CLF: A Dependent Logical Framework for Concurrent Computations*

Kevin Watkins
Carnegie Mellon University
kw@cs.cmu.edu

Iliano Cervesato
ITT Industries
iliano@itd.nrl.navy.mil

Frank Pfenning
Carnegie Mellon University
fp@cs.cmu.edu

David Walker
Princeton University
dpw@cs.princeton.edu

Abstract

We present CLF, a dependently typed logical framework with several novel features supporting concurrent computations, in particular monads and synchronous linear connectives. We illustrate its representation methodology of *concurrent computations as monadic expressions* via the encoding of an asynchronous π-calculus with correspondence assertions, including its dynamic semantics, safety criterion, and a type system with latent effects due to Gordon and Jeffrey. We also explain a new, general methodology for defining dependently-typed logical frameworks in the LF family. The methodology involves defining the terms of the framework directly in canonical form and greatly simplifies much of the metatheory of these frameworks, which has been notoriously difficult in the past. We have used the methodology to show that CLF has a number of key properties including decidability of type checking.

1 Introduction

Logical frameworks are meta-languages that allow users to represent deductive systems such as type systems, operational semantics, logics and proofs. A logical framework is usually characterized by two distinct elements: the language itself, be it a first-order logic or a type theory, and the representation methodology that users employ to encode the deductive systems of interest within the language.

When it comes to the design and analysis of programming languages in particular, logical frameworks can serve two related functions. On the one hand, when the goal is to represent and reason about a pre-existing language, a logical framework serves as a tool for encoding the abstract syntax and static and dynamic semantics of the language in question. Once these basic judgments have been faithfully encoded in the framework, one can further characterize their properties in the framework itself. For instance, a natural next step in an analysis of a type-safe programming language is to write down a predicate that specifies the safety property for program executions.

On the other hand, when the goal is to develop a new language or reconsider the semantics of an old one, the framework itself may partly suggest a language definition or its concise formulation. In other words, our understanding of the features of a framework and its representation techniques can help us engineer a precise and principled definition for a language. Often, the presentation of a language definition that comes from working within a logical framework in this way is simpler and more uniform than it otherwise might have been.

*This research was sponsored in part by the NSF under grants CCR-9988281, CCR-0208601, CCR-0238328, and CCR-0306313, and by NRL under grant N00173-00-C-2086.

In order to serve these two functions well, a framework must be simple and uniform, yet provide support for representing common concepts in its application domain. For instance, the logical framework LF [HHP93] is built around the single notion of dependent functions. Dependent functions capture two of the most pervasive concepts in programming languages, the notions of hypothetical and parametric judgments. Consequently, dependent functions alone make LF an excellent framework for representing pure languages and logics. Unfortunately, since all hypotheses in LF are subject to weakening and contraction, representing imperative program state is complex and cumbersome and unlikely to inform programming language design. To provide support for simple state changes, the framework LLF [CP02] augments LF with three connectives from linear logic, namely $\multimap$, & and $\top$. The LLF representation methodology suggests representing state as linear hypotheses, and state-changing operations as linear functions.

This paper develops the dependently typed framework CLF, which is a conservative extension of LF and LLF with intrinsic support for the representation of concurrent languages. This support comes in the form of the additional linear connectives !, $\exists$, $\otimes$ and 1. Each of these connectives helps support simple and natural representations of concurrent computations. However, on their own, they are incompatible with the LF or LLF type theory. In fact, adding even a single one of these connectives to LF immediately destroys many crucial properties of framework and invalidates the central representation methodology! To solve this problem, we have adapted an old idea from programming language researchers to the world of logical frameworks: We separate the well-behaved LLF fragment of our framework from our new fragment by encapsulating the latter inside a monad [Mog91].

The monadic expressions we introduce form the basis for representing concurrent computations. Intuitively, these monadic expressions are a sequence of computational steps consuming some linear hypotheses and assuming others, thereby changing the state. If one step introduces a hypothesis that is consumed in a second step, then the second step depends on the first one; otherwise steps are independent. In order to move from a framework that represents simple sequential state change to one that captures *true concurrency*, the framework identifies any two monadic expressions that differ only in the order of independent steps. Since monadic expressions are first-class objects in the framework, programmers can then use the framework itself to specify properties of concurrent executions via predicates and relations over monadic expressions.

We illustrate the framework's expressive power and representation techniques through a sample encoding of an asynchronous version of the π-calculus with correspondence assertions, following Gordon and Jeffrey [GJ03]. The CLF encoding of its dynamic semantics yields monadic expressions that correspond exactly to its legal computations. Dependent types indexed by these expressions formalize the correspondence property for execution, which is the relevant safety criterion. We also formalize a type system with latent effects akin to Gordon and Jeffrey's. While not a significant contribution to the theory of correspondence assertions, this work nonetheless sheds some interesting light on it. For instance, the operational semantics we give appears somewhat simpler than the original specification given by Gordon and Jeffrey because it avoids development of a relatively complex process equivalence relation. We also avoid a labeled transition system, instead specifying the correspondence policy directly as a predicate on program executions (a type family indexed by monadic expressions in the framework). We arrived at these definitions by experimenting with multiple variations within the new framework.

We have had a similar experience on other examples we have investigated as well. For instance, while encoding the semantics of Concurrent ML we developed a new style of presenting the operational semantics which is more modular than the published specification [Rep99]. The Concurrent ML example and a number of others may be found in an extensive technical report on CLF [CPWW02].

Another significant technical contribution of this paper is our methodology for defining dependently-typed languages in the LF family. In particular, we define CLF and its type-checking algorithm directly on the canonical forms of objects in the framework. Our techniques greatly simplify the meta-theory of dependently-typed languages, which has been extremely difficult even for much simpler dependent languages such as LF alone. In this short paper, we outline this methodology and present some of the most important meta-theoretic results. A second technical report [WCPW02] contains proofs of all the theorems we present.

In earlier work, we presented the propositional fragment of CLF [WCPW04]. The current paper represents a substantial leap forward as it integrates these earlier ideas within a framework containing dependent and existential types and indexed type families. More specifically, in the propositional fragment, it is possible to encode the operational semantics of Petri Nets, the simplest of concurrent languages, but dependent and existential types are necessary for representing the operational semantics of almost any other programming language and any type system. Moreover, in the propositional fragment, monadic expressions cannot be manipulated as objects, and therefore, it is not possible to analyze the properties of concurrent computations. In this paper, we show how to specify properties of concurrent computations, such as the safety property for correspondence assertions, by defining type families indexed by monadic expressions. Finally, the meta-theory necessary to develop the propositional fragment of the logic is almost trivial when compared with the meta-theory necessary to handle dependency. In particular, the central difficulty in defining our framework based on canonical forms is how to handle the elimination form for dependent functions. This elimination form demands substitution of one canonical object within another, which, if done naively, will yield a non-canonical result. To solve this problem, we present an algorithm that simultaneously substitutes and normalizes objects and terminates even on terms that are not well-typed. This algorithm is defined by nested induction on the structure of a type and a term.

2 The unrestricted connectives

At CLF's core is the type theory λ^{Π} of the framework LF. The theory consists of type constructors, which may have dependent function kinds $\Pi u\!:\!A.\,K$ and objects, which may have dependent function types $\Pi u\!:\!A.\,B$. The type constructors represent judgments concerning the syntax, the static semantics and other features of language; the objects represent proofs that particular judgments are valid.

In order to be effective, a logical framework in this style must possess an algorithm for checking that objects are well-typed and consequently that particular judgements the user has encoded are valid. In the past, proving the decidability of algorithms for type-checking LF-style frameworks has been a very complex task, and despite being experts in this area, we would have found it extremely difficult to scale this proof up sufficiently to handle a language like CLF. Fortunately, we have found a novel technique for defining these dependent frameworks that simplifies the task of proving decidability tremendously.

The first step in our new methodology is to adopt a syntax in which all objects are canonical (β-normal and η-long), syntactically ruling out redices. The introduction and elimination forms (from the point of view of logic) become separate syntactic classes. We call them *atomic* and *normal* objects, respectively. Likewise, there are atomic and normal types (type *families*, strictly speaking), and normal kinds. These syntactic constructs and associated contexts are as follows. (While we focus on the λ^{Π} sublanguage, ellipses indicate that there is more to come):

Normal kinds	K	$::= \mathsf{type} \mid \Pi u\!:\!A.\,K$
Atomic types	P	$::= a \mid P\,N$
Normal types	A	$::= P \mid \Pi u\!:\!A_1.\,A_2 \mid \ldots$
Atomic objects	R	$::= c \mid u \mid R\,N \mid \ldots$
Normal objects	N	$::= R \mid \lambda u.\,N \mid \ldots$
Unrestricted contexts	Γ	$::= \cdot \mid \Gamma, u\!:\!A$
Signatures	Σ	$::= \cdot \mid \Sigma, a\!:\!K \mid \Sigma, c\!:\!A$

The metavariable c stands for constants, while u stands for variables. These take their types from a *signature* and a (typing) *context*, respectively. The type family constants a also take their kinds from the signature. As usual, we only consider terms modulo the renaming of bound variables and maintain that constants and variables are declared at most once in a signature or context, respectively. We often write $A \to B$ instead of $\Pi u\!:\!A.\,B$ or $A \to K$ instead of $\Pi u\!:\!A.\,K$ when B or K, respectively, contains no free occurrence of u. Also, it is clearer in some cases to write $B \leftarrow A$ for $A \to B$.

Typing for canonical forms is pleasantly simple. Each atomic object (or type) has a unique type (or kind) in a given context. Each normal object (or type) can be checked for well-formedness, given its putative type (or kind). The structure of a normal object is constrained by its type; it is in this sense that they are canonical forms. The notation for these judgments is as follows (the context Δ may be ignored prior to Section 3):

N is normal of type A	$\Gamma;\Delta \vdash_\Sigma N \Leftarrow A$
R is atomic of type A	$\Gamma;\Delta \vdash_\Sigma R \Rightarrow A$

In each case, Γ, Δ, and Σ are known inputs to the judgment and the arrow points in the direction information flows:

$$
\begin{array}{rcl}
\mathsf{stop} & : & \mathsf{pr}. \\
\mathsf{par} & : & \mathsf{pr} \to \mathsf{pr} \to \mathsf{pr}. \\
\mathsf{repeat} & : & \mathsf{pr} \to \mathsf{pr}. \\
\mathsf{new} & : & \mathsf{tp} \to (\mathsf{nm} \to \mathsf{pr}) \to \mathsf{pr}. \\
\mathsf{choose} & : & \mathsf{pr} \to \mathsf{pr} \to \mathsf{pr}. \\
\mathsf{out} & : & \mathsf{nm} \to \mathsf{nm} \to \mathsf{pr}. \\
\mathsf{inp} & : & \mathsf{nm} \to \mathsf{tp} \to (\mathsf{nm} \to \mathsf{pr}) \to \mathsf{pr}. \\
\mathsf{begin} & : & \mathsf{label} \to \mathsf{pr} \to \mathsf{pr}. \\
\mathsf{end} & : & \mathsf{label} \to \mathsf{pr} \to \mathsf{pr}.
\end{array}
\qquad
\begin{array}{rcl}
\ulcorner \mathsf{stop} \urcorner & = & \mathsf{stop} \\
\ulcorner P \mid Q \urcorner & = & \mathsf{par}\ \ulcorner P \urcorner\ \ulcorner Q \urcorner \\
\ulcorner \mathsf{repeat}\ P \urcorner & = & \mathsf{repeat}\ \ulcorner P \urcorner \\
\ulcorner \mathsf{new}(x\!:\!\tau); P \urcorner & = & \mathsf{new}\ \ulcorner \tau \urcorner\ (\lambda x. \ulcorner P \urcorner) \\
\ulcorner \mathsf{choose}\ P\ Q \urcorner & = & \mathsf{choose}\ \ulcorner P \urcorner\ \ulcorner Q \urcorner \\
\ulcorner \mathsf{out}\ x\langle y \rangle \urcorner & = & \mathsf{out}\ x\ y \\
\ulcorner \mathsf{inp}\ x(y\!:\!\tau); P \urcorner & = & \mathsf{inp}\ x\ \ulcorner \tau \urcorner\ (\lambda y. \ulcorner P \urcorner) \\
\ulcorner \mathsf{begin}\ L; P \urcorner & = & \mathsf{begin}\ \ulcorner L \urcorner\ \ulcorner P \urcorner \\
\ulcorner \mathsf{end}\ L; P \urcorner & = & \mathsf{end}\ \ulcorner L \urcorner\ \ulcorner P \urcorner
\end{array}
$$

Figure 1: CLF representation of π-calculus syntax

in the first judgment, the type A is an input that is used to check a normal object; in the second judgment, the type A is an output that is synthesized from an atomic object. Often Σ is omitted, as the signature is constant throughout any typing derivation. All of the typing rules are syntax directed: there are no structural rules or type conversion rules that may be applied at arbitrary points in the derivation, and the inputs necessary to decide each premise of a rule are completely determined either by inputs to the conclusion or by outputs of premises to the left. (In other words, we are implicitly defining a primitive recursive decision procedure for typing, not merely a potentially undecidable judgment.) The rules for constants, variables, introduction of dependent functions and equality illustrate these ideas:

$$
\frac{}{\Gamma; \Delta \vdash c \Rightarrow \Sigma(c)}\ c
\qquad
\frac{}{\Gamma; \Delta \vdash x \Rightarrow \Gamma(x)}\ x
$$

$$
\frac{\Gamma, u\!:\!A; \Delta \vdash N \Leftarrow B}{\Gamma; \Delta \vdash \lambda u.\, N \Leftarrow \Pi u\!:\!A.\, B}\ \Pi\mathbf{I}
$$

$$
\frac{\Gamma; \Delta \vdash R \Rightarrow P' \quad P' = P}{\Gamma; \Delta \vdash R \Leftarrow P}\ \Rightarrow\Leftarrow
$$

The checking of a λ-abstraction $\lambda u.\, N$ at type $\Pi u\!:\!A.\, B$ reduces by $\Pi\mathbf{I}$ to the checking of its body N in a context extended by u:A. The checking of an atomic object R at type P reduces by $\Rightarrow\Leftarrow$ to the synthesis of a type P' for R and the verification that P and P' are equal. For the λ^{Π} fragment, the equality $P = P'$ is simply α-convertibility. Since P is a metavariable ranging over atomic types, only η-long forms will typecheck. Finally, the rules for constants and variables allow their types to be synthesized directly from the context or signature.

In order to complete the rules for typing λ^{Π}, we need to assign a type synthesis rule to the application $R\ N$ of dependent functions. Normally, one would expect a function R having dependent type $\Pi u\!:\!A.\, B$, applied to an argument N of type A, to synthesize the type $[N/u]B$, where $[N/u]$ is metasyntax for the ordinary capture-avoiding substitution of N for u. In our syntax, the ordinary substitution cannot even be defined, since variables u are atomic while N is normal, and there is no coercion from normal to atomic objects (which would create β-redices).

Felty's canonical LFaddresses this issue by introducing an additional syntax for non-canonical terms, and defining the usual notion of β-reduction on this syntax. Then the typing rule for application synthesizes the normal form (which can be shown to be canonical) of the non-canonical substitution $[N/u]B$. A difficulty with this approach is that reduction is inextricably intertwined with typing, and so the subject reduction property and strong normalization property are quite tricky to establish. (Indeed, Felty does not do so directly, instead appealing implicitly to subject reduction and strong normalization with respect to the original type system for LF and introducing another argument showing her canonical type system to be equivalent. Of course, the canonical type system is then dependent on a preexisting type system for non-canonical terms, with its own tricky metatheory, and cannot be defined *ab initio*.)

We improve on Felty's approach by observing that the process of reduction of a substitution $[N/u]B$ to canonical form can be defined by higher-order primitive recursion on the type of the variable u. We denote this primitive recursive functional $\mathsf{inst\_a}_A(u.\, B, N)$, where A is the type of the variable u, and call it *instantiation*. The primitive recursion terminates regardless of whether the terms B and N are well-typed. This makes it trivial that typing is decidable (indeed, higher-order primitive recursive), and a "subject reduction" theorem for instantiation can be established by elementary inductive arguments.

With the new instantiation operator in hand, the elimination form for dependent functions has the following structure.

$$
\frac{\Gamma; \Delta \vdash R \Rightarrow \Pi u\!:\!A.\, B \quad \Gamma; \cdot \vdash N \Leftarrow A}{\Gamma; \Delta \vdash R\ N \Rightarrow \mathsf{inst\_a}_A(u.\, B, N)}\ \Pi\mathbf{E}
$$

There are also well-formedness rules for kinds and types, posing no additional difficulties—for details, consult Appendix A.

Although our presentation of λ^{Π} differs from that of the original LF paper, it can be shown to be equivalent in the strong sense that the canonical form of any LF term is well-typed in the canonical system, and any typing derivation in the canonical system maps to an isomorphic derivation in the original type system for LF.

We take care to ensure that as the λ^{Π} fragment is extended to full CLF, it remains conservative over LF. Indeed, the key principles for constructing representations in LF, *judgments as types* and *higher-order abstract syntax*, still apply.

2.1 An introduction to the asynchronous π-calculus with correspondence assertions

To illustrate the various features of CLF, we will develop a running example involving the asynchronous π-calculus with embedded *correspondence assertions* [WL93]. As we introduce elements of CLF we will show how to use them by representing the syntax of the π-calculus constructs, a type system for checking correspondences, based on work by Gordon and Jeffrey [GJ03], a dynamic semantics for the language and finally a specification of the correspondence property (which must hold for an execution in the language to be safe).

Correspondence assertions, originally developed by Woo and Lam [WL93], allow some safety properties of concurrent programs to be verified by marking significant points in a program with assertions $\mathsf{begin}\ L$ or $\mathsf{end}\ L$, where L is a *label* carrying information about the state of the program. An execution is safe if it satisfies the following *correspondence property*: for each $\mathsf{end}\ L$ assertion reached in an execution, a distinct $\mathsf{begin}\ L$ assertion (for the same L) must have been reached in the past. By choosing carefully where to place correspondence assertions, interesting safety properties can be reduced to the correspondence property. Woo and Lam, and Gordon and Jeffrey, have shown how to do this for a variety of important correctness properties of cryptographic protocols.

To illustrate the basic ideas, we will examine an extremely simple handshake protocol taken directly from Gordon and Jeffrey's work. We wish to send a message reliably from a process a to another process b. When b receives the original message, it returns an acknowledgment to a. The protocol is intended to ensure that if a receives an acknowledgment message then b actually did receive the original message. In the asynchronous π-calculus with correspondence assertions, we specify the protocol as follows.

$$\begin{aligned}\mathrm{Send}(a,b,c) &= \mathsf{new}(msg); \mathsf{new}(ack);\\ &\qquad (\mathsf{out}\ c\langle msg, ack\rangle\\ &\qquad\ \mid \mathsf{inp}\ ack(); \mathsf{end}\ (a,b,msg); \mathsf{stop})\\ \mathrm{Rcv}(a,b,c) &= \mathsf{inp}\ c(msg, ack); \mathsf{begin}\ (a,b,msg); \mathsf{out}\ ack\langle\rangle\end{aligned}$$

The standard π-calculus process constructors used here are the parallel composition $P \mid Q$, the do-nothing process stop, the name binding operator $\mathsf{new}(x); p$ (where a new name x is bound for use in P), the asynchronous output operator $\mathsf{out}\ c\langle x_1, \ldots, x_n\rangle$, and the asynchronous input operator $\mathsf{inp}\ c(x_1, \ldots, x_n); P$ (where variables x_1 through x_n are bound in P).

In the specification above, the sending process a generates a new message msg and a new acknowledgment channel ack. The sender uses the asynchronous output operator to send them as a pair on the (already established) channel c, and waits for a response on ack. Once the sender receives the acknowledgment, it executes an end assertion with the label (a, b, msg). The semantics of the label is that the sender a requires that the receiver b has already received the input message msg.

The receiver cooperates with the sender by waiting for pairs of message and acknowledgment channel on channel c. After receiving on c, a begin assertion declares that the receiver b has received the input message. After this declaration, the receiver sends an acknowledgment to the sender. Safety requires that in all executions of senders in parallel with receivers, end assertions have matching begin assertions. If so, sender a can be sure that receiver b received the message msg.

Now, consider combining a single sender in parallel with a single receiver: $\mathsf{new}(c); (\mathrm{Send}(a,b,c) \mid \mathrm{Rcv}(a,b,c))$. This configuration is *safe* since in every possible execution, every $\mathsf{end}\ (a,b,msg)$ assertion is preceded in that execution by a distinct corresponding $\mathsf{begin}\ (a,b,msg)$ assertion. On the other hand, placing multiple different senders in parallel with a single copy of a receiver is *unsafe*:

$$\mathsf{new}(c); (\mathrm{Send}(a,b,c) \mid \mathrm{Send}(a',b,c) \mid \mathrm{Rcv}(a,b,c))$$

This configuration is unsafe because there exists an execution in which an $\mathsf{end}\ L$ assertion is executed but there has been no prior matching $\mathsf{begin}\ L$. More specifically, the second sender a' may create a message and send it to the receiver. The receiver, thinking it is communicating with a, receives the message, executes $\mathsf{begin}\ (a,b,msg)$, and returns the acknowledgment. Finally, the second sender executes $\mathsf{end}\ (a',b,msg)$. In this protocol, since the identity of the sender (either a or a') is not included in the message, there can be confusion over whom the receiver is communicating with. While this example is very simple, Gordon and Jeffrey have demonstrated that these assertions can be used to identify flaws in more complex protocols as well [GJ03].

2.2 Representing syntax

The first component of our CLF representation of the π-calculus is a representation of its syntax, following standard LF methodology. For simplicity, our π-calculus contains monadic input and output processes rather than polyadic ones, and the only data structures are names x, y, z. We also removed the if-then-else construct of Gordon et al. in favor of a non-deterministic choice operator.[1]

Two process forms that did not show up in the informal example in Section 2.1 are the replicated process $\mathsf{repeat}\ P$, which acts as an unbounded number of copies of P, and the non-deterministic choice operator $\mathsf{choose}\ P\ Q$. The syntax refers to types τ of the static semantics, which will be discussed later. We do not specify any particular syntax for assertion labels L, but it is assumed that they might mention names bound by new or inp. As usual, bound names are allowed to α-vary without explicit mention of α-conversion. Channels are a special case of names.

$$\begin{aligned}P, Q \quad ::= &\ \mathsf{stop} \mid (P \mid Q) \mid \mathsf{repeat}\ P \mid \mathsf{new}(x{:}\tau); P\\ &\mid \mathsf{choose}\ P\ Q \mid \mathsf{out}\ x\langle y\rangle \mid \mathsf{inp}\ x(y{:}\tau); P\\ &\mid \mathsf{begin}\ L; P \mid \mathsf{end}\ L; P\end{aligned}$$

The corresponding LF signature, shown on the left of Figure 1, represents process syntax via CLF types pr (processes), nm (names), tp (types), and label (assertion labels). The representation function mapping processes to CLF objects, written $\ulcorner\_\urcorner$, is shown at the right.

A few comments: The type nm of names does not contain any closed terms; it classifies bound variables within a process expression. The type tp is discussed in Section 3. As is common in LF representations, we use *higher-order abstract syntax*, which allows us to model π-calculus bound variables using framework variables and to implement π-calculus substitution using the framework's substitution.

The most important property of this representation is *adequacy*: every process in the original language has its own representative as a CLF object of type pr, and every object in pr is such a representation. The canonical forms property for CLF renders proofs of such properties trivial.

3 The asynchronous linear connectives

The next larger sublanguage contained in CLF is the type theory $\lambda^{\Pi\multimap\&\top}$, the basis of Cervesato and Pfenning's LLF (Linear Logical Framework) [CP02]. $\lambda^{\Pi\multimap\&\top}$ extends λ^{Π}

[1] The if-then-else construct, or guarded process, requires the ability to decide whether names are equal. Guards testing equality of names are easy to represent; those testing disequality of names less so. This difficulty in characterizing disequality of names is common to the whole lineage of frameworks based on LF.

$$
\begin{array}{rcl}
\mathsf{good} & : & \mathsf{pr} \to \mathsf{type}. \\
\mathsf{consume} & : & \mathsf{eff} \to \mathsf{type}. \qquad \textit{\% See Section 5} \\
\mathsf{assume} & : & \mathsf{eff} \to \mathsf{pr} \to \mathsf{type}. \ \textit{\% See Section 5} \\
\\
\mathsf{gd\_stop} & : & \mathsf{good\ stop} \multimapinv \top. \\
\mathsf{gd\_par} & : & \mathsf{good}\ (\mathsf{par}\ P\ Q) \multimapinv \mathsf{good}\ P \multimapinv \mathsf{good}\ Q. \\
\mathsf{gd\_repeat} & : & \mathsf{good}\ (\mathsf{repeat}\ P) \multimapinv \top \leftarrow \mathsf{good}\ P. \\
\mathsf{gd\_new} & : & \mathsf{good}\ (\mathsf{new}\ \tau\ (\lambda x.\, P\ x)) \multimapinv (\Pi x{:}\mathsf{nm}.\, \mathsf{has}\ x\ \tau \to \mathsf{good}\ (P\ x)). \\
\mathsf{gd\_choose} & : & \mathsf{good}\ (\mathsf{choose}\ P\ Q) \multimapinv (\mathsf{good}\ P \,\&\, \mathsf{good}\ Q). \\
\mathsf{gd\_out} & : & \mathsf{good}\ (\mathsf{out}\ X\ Y) \leftarrow \mathsf{has}\ X\ (\mathsf{chan}\ \tau\ (\lambda y.\, E\ y)) \leftarrow \mathsf{has}\ Y\ \tau \multimapinv \mathsf{consume}\ (E\ Y). \\
\mathsf{gd\_inp} & : & \mathsf{good}\ (\mathsf{inp}\ X\ \tau\ (\lambda y.\, P\ y)) \leftarrow \mathsf{has}\ X\ (\mathsf{chan}\ \tau\ (\lambda y.\, E\ y)) \\
 & & \quad \leftarrow (\Pi y{:}\mathsf{nm}.\, \mathsf{has}\ y\ \tau \to \mathsf{assume}\ (E\ y)\ (P\ y)). \\
\mathsf{gd\_begin} & : & \mathsf{good}\ (\mathsf{begin}\ L\ P) \multimapinv (\mathsf{effect}\ L \multimap \mathsf{good}\ P). \\
\mathsf{gd\_end} & : & \mathsf{good}\ (\mathsf{end}\ L\ P) \multimapinv \mathsf{effect}\ L \multimapinv \mathsf{good}\ P.
\end{array}
$$

Figure 2: Static semantics represented in CLF

with linear hypotheses and as many connectives of intuitionistic linear logic as have invertible introductions (a property intimately connected to the strong canonical forms property). Such connectives have been called *asynchronous* by Andreoli [And92]. Following this terminology, we refer to the λ^{Π} types together with the new connectives as the *asynchronous* types. We do not consider linear dependent functions like RLF's [IP98] (even in full CLF).

The additional syntax is as follows:

$$
\begin{array}{lll}
\text{Asynch. types} & A & ::= \ldots \mid A_1 \multimap A_2 \mid A_1 \,\&\, A_2 \mid \top \mid \ldots \\
\text{Atomic objects} & R & ::= \ldots \mid x \mid R\,\hat{}\,N \mid \pi_1 R \mid \pi_2 R \mid \ldots \\
\text{Normal objects} & N & ::= \ldots \mid \hat{\lambda} x.\, N \mid \langle N_1, N_2 \rangle \mid \langle\rangle \mid \ldots \\
\\
\text{Linear contexts} & \Delta & ::= \cdot \mid \Delta, x \hat{:} A
\end{array}
$$

The linear function type $A_1 \multimap A_2$ (sometimes also written $A_2 \multimapinv A_1$) introduces linear hypotheses, which we collect in a separate context Δ. We implicitly regard Δ as an unordered multiset, because unlike unrestricted hypotheses, linear hypotheses cannot depend on one other. We consider the two sorts of variable u and x to be in different syntactic categories, but in practice we do not distinguish between them. The typing rules for linear functions, and for the additive product & and unit $\top$ as well, are type-theoretic generalizations of the corresponding rules of linear logic:

$$
\frac{}{\Gamma; \Delta \vdash \langle\rangle \Leftarrow \top}\ \top\mathbf{I}
\qquad
\frac{\Gamma; \Delta, x \hat{:} A \vdash N \Leftarrow B}{\Gamma; \Delta \vdash \hat{\lambda} x.\, N \Leftarrow A \multimap B}\ {\multimap}\mathbf{I}
$$

$$
\frac{\Gamma; \Delta_1 \vdash R \Rightarrow A \multimap B \quad \Gamma; \Delta_2 \vdash N \Leftarrow A}{\Gamma; \Delta_1, \Delta_2 \vdash R\,\hat{}\,N \Rightarrow B}\ {\multimap}\mathbf{E}
$$

$$
\frac{\Gamma; \Delta \vdash N_1 \Leftarrow A \quad \Gamma; \Delta \vdash N_2 \Leftarrow B}{\Gamma; \Delta \vdash \langle N_1, N_2 \rangle \Leftarrow A \,\&\, B}\ \&\mathbf{I}
$$

$$
\frac{\Gamma; \Delta \vdash R \Rightarrow A \,\&\, B}{\Gamma; \Delta \vdash \pi_1 R \Rightarrow A}\ \&\mathbf{E}_1
\qquad
\frac{\Gamma; \Delta \vdash R \Rightarrow A \,\&\, B}{\Gamma; \Delta \vdash \pi_2 R \Rightarrow B}\ \&\mathbf{E}_2
$$

(There is no elimination rule for $\top$.)

A major benefit of linear hypotheses is the ability to represent *stateful systems* [CP02] using the methodology associated with Cervesato and Pfenning's LLF. In our π-calculus example, we will be able to represent the static "state" of the type and effect system for correspondence assertions as a set of linear hypotheses in LLF style. The basic idea is to record a multiset of begins already reached at the current program point as linear hypotheses of the typing judgment.[2] Then each occurrence of begin L contributes a linear hypothesis of type effect L for the checking of its continuation, and each end L consumes such a hypothesis.

3.1 Representing a type and effect system

To illustrate the use of CLF's linear connectives, we will encode a variant of Gordon and Jeffrey's type system with latent effects [GJ03]. The goal of the static semantics is to define a decidable sufficient condition ensuring that the correspondence property is not violated in any possible execution of a program. (Recall that the correspondence property requires that each end L in an execution be preceded by a distinct begin L for the same label L.) To do this, the static semantics associates an effect Λ (a multiset of labels) with each program point, such that it is safe to execute end L for each label L in the multiset. The typing rule for begin $L; P$ adds L to the effect for checking the continuation P, while the typing rule for end L removes such an L. (Of course, not all safe programs will necessarily have a valid typing.)

This typing discipline accounts for trivial instances of correct programs in which an end is found directly within the continuation of its matching begin. Of course, in actual use, one is more interested in cases in which the end and its matching begin occur in different processes executing concurrently (as in the example of Section 2.1).

Gordon et al. introduce *latent effects* to treat many such cases. The idea is that each value transmitted across a channel may carry with it a multiset e of latent effects, the effects being *debited* from the process sending the value and *credited* to the process receiving it. Since communication temporally orders the sending and receiving processes, it is certain that the begins introducing the debited effects in the sending process will occur before any ends making use of the credited effects in the receiving process.[3]

These considerations lead to a simple type syntax. Each name in the static semantics has a type τ: either Name (e.g., a nonce) or $\mathsf{Ch}(x{:}\tau)e$, representing a channel transmitting

[2] Really these are *affine* hypotheses, since the invariant is that the multiset be merely a lower bound: it is perfectly safe to "forget" that a begin was reached at some point in the past. Careful use of the additives $\top$ and & will allow us to simulate affine hypotheses with linear ones.

[3] Of course, this implicitly relies on the unicast nature of communication in the language. If multicast or broadcast were allowed, more than one process could be credited, violating the non-duplicable nature of effect hypotheses.

names of type τ and a latent effect e. Although we have left the syntax of labels unspecified, in applications they will often depend on names; thus, the occurrence of x in $\mathsf{Ch}(x\!:\!\tau)e$ is a binder for any free occurrences of x in e, and it α-varies in the usual way. We write Λ for a multiset of labels, and Λ_1, Λ_2 for a disjoint union of multisets.

$$\begin{array}{rl} \tau & ::= \mathsf{Name} \mid \mathsf{Ch}(x\!:\!\tau)e \\ e & = [\Lambda] \\ \Theta & ::= \cdot \mid \Theta, x\!:\!\tau \\ \Lambda & ::= \cdot \mid \Lambda, L \end{array}$$

In the framework, types τ are represented as objects with CLF type tp. The representation of latent effects e is discussed in Section 5 below; for present purposes, assume that effects will be represented as objects with type eff. A typing context Θ will be represented by a collection of unrestricted assumptions ($\mathsf{has}\ X\ T$) in the CLF context and an effect context Λ will be represented by a series of linear assumptions ($\mathsf{effect}\ L$) in the CLF context. Neither the type family has nor the type family effect contains any closed objects.

$$\begin{array}{rcl} \mathsf{tp} & : & \mathsf{type}. \\ \mathsf{name} & : & \mathsf{tp}. \\ \mathsf{chan} & : & \mathsf{tp} \to (\mathsf{nm} \to \mathsf{eff}) \to \mathsf{tp}. \\[4pt] \mathsf{has} & : & \mathsf{nm} \to \mathsf{tp} \to \mathsf{type}. \\ \mathsf{effect} & : & \mathsf{label} \to \mathsf{type}. \end{array}$$

We first present the static semantics of the language as an inference system. Informally, the main typing judgment $\Theta; \Lambda \vdash P$ asserts that the process P will safely execute in a context given by names Θ and effect Λ, which we think of as a sort of capability for executing $\mathsf{end}\ L$s.

$$\frac{}{\Theta; \Lambda \vdash \mathsf{stop}} \qquad \frac{\Theta; \Lambda_1 \vdash P \quad \Theta; \Lambda_2 \vdash Q}{\Theta; \Lambda_1, \Lambda_2 \vdash P \mid Q}$$

$$\frac{\Theta; \cdot \vdash P}{\Theta; \Lambda \vdash \mathsf{repeat}\ P} \qquad \frac{\Theta, x\!:\!\tau; \Lambda \vdash P}{\Theta; \Lambda \vdash \mathsf{new}(x\!:\!\tau); P}$$

$$\frac{\Theta; \Lambda \vdash P \quad \Theta; \Lambda \vdash Q}{\Theta; \Lambda \vdash \mathsf{choose}\ P\ Q}$$

$$\frac{(x\!:\!\mathsf{Ch}(z\!:\!\tau)[\Lambda_1]) \in \Theta \quad (y\!:\!\tau) \in \Theta}{\Theta; [y/z]\Lambda_1, \Lambda_2 \vdash \mathsf{out}\ x\langle y \rangle}$$

$$\frac{(x\!:\!\mathsf{Ch}(y\!:\!\tau)[\Lambda_2]) \in \Theta \quad \Theta, y\!:\!\tau; \Lambda_1, \Lambda_2 \vdash P}{\Theta; \Lambda_1 \vdash \mathsf{inp}\ x(y\!:\!\tau); P}$$

$$\frac{\Theta; \Lambda, L \vdash P}{\Theta; \Lambda \vdash \mathsf{begin}\ L; P} \qquad \frac{\Theta; \Lambda \vdash P}{\Theta; \Lambda, L \vdash \mathsf{end}\ L; P}$$

The rules for new and inp carry the proviso that x or y, respectively, not occur free in the conclusion of an instance of each rule.

Next, in the framework, we represent the π-calculus typing judgment as a CLF type family good, shown in Figure 2. The type A in $\Pi u\!:\!A.\,B$ has been omitted where it is determined by context. We often omit outermost Π quantifiers entirely; in such cases the corresponding arguments to the constant in question are also omitted (implicit). We have also η-contracted some subterms to conserve space; these should be read as abbreviations for their η-long (canonical) forms.

Since not every declared effect must actually occur (that is, effects are *affine* and implicitly admit a *weakening* principle), we must use the additive unit $\top$ to consume any leftover effects at the leaves of a derivation (instances of the $\mathsf{gd\_stop}$ rule) and where the effect set is reset to empty (instances of the $\mathsf{gd\_repeat}$ rule).

The task of assume and $\mathsf{consume}$ is to introduce and consume linear hypotheses for the whole multiset of effects contained in a latent effect. Latent effects are consumed by out, which has no continuation, and produced by inp, which does. Accordingly, assume takes the continuation as an argument, and invokes good to check it once the multiset of effects has been introduced into the linear context. The representations of assume and $\mathsf{consume}$ are shown in Section 5.

It can be shown that this representation is *adequate*: a process P is well-typed in the original system just when there is an object of type $\mathsf{good}\ P$ in CLF.

4 The lax modality and synchronous connectives

Thus far we have seen the connectives CLF inherits from earlier frameworks. It is time now to introduce the syntax for concurrent computations that is the raison d'être of the framework. The principal challenge lies in the need to retain conservativity over $\lambda^{\Pi\multimap\&\top}$: because the structure of a concurrent computation will not necessarily be determined by its type, there is the danger that the strong canonical forms property that $\lambda^{\Pi\multimap\&\top}$ enjoys could be lost.[4]

Fortunately, the term language of *judgmental lax logic* [PD01] provides a ready-made solution. In addition to the normal judgment $N \Leftarrow A$, which may be thought of as expressing the truth of A, it has a second judgment $E \leftarrow A$, where E is a new syntactic class of *expressions*. In computational terms, the new judgment distinguishes *effectful computations* from the *effect-free values* of the original judgment $N \Leftarrow A$. (Here we think of the inherent nondeterminism of concurrent computations as an effect.)

Since "possibly effectful" is weaker than "effect-free," there is an inclusion of the normal objects N into the expressions E. There is no reverse inclusion; instead, the notion of "possibly effectful" is internalized by a new *monadic type constructor* $\{\cdot\}$, so called because it satisfies the axioms of a monad.

Next, what synchronous types S should be available? In other words, what kinds of result can a computation have? We treat three related phenomena: *fresh name generation*, the creation of *new unrestricted hypotheses*, and the creation of *multiple related linear hypotheses*. Each of these corresponds to a synchronous connective of intuitionistic linear logic, in Andreoli's terminology [And92].

Concerning fresh name generation, a plausible idea might be to model a computation generating a name x of type A by an *existential type*: $\exists u\!:\!A.\,B$. However, we must be careful: the elimination rule for $\exists$ would destroy the strong canonical forms property enjoyed by $\lambda^{\Pi\multimap\&\top}$. Fortunately, for our purposes such an elimination rule is needed only in the *lax* (monadic) judgment, since $\exists$ is only to be used in computations. The creation of multiple related linear hypotheses is modeled by a multiplicative conjunction $\otimes$ and unit 1, and

[4] See our technical report [WCPW02] for more discussion of this subtle point. Also, though it is not treated here, the planned logic programming interpretation of CLF relies on the uniform proofs property, which is connected to the strong canonical forms property.

the creation of new unrestricted hypotheses is modeled by the unrestricted modality !.

These ideas lead to the following (final) syntax for CLF. We represent this type theory symbolically as $\lambda^{\Pi\multimap\&\top\{\exists\otimes 1!\}}$.

Asynchronous types	A	$::= \ldots \mid \{S\}$
Normal objects	N	$::= \ldots \mid \{E\}$
Expressions	E	$::= \mathsf{let}\ \{p\} = R\ \mathsf{in}\ E \mid M$
Synchronous types	S	$::= A \mid \exists u\!:\!A.\,S \mid S_1 \otimes S_2 \mid 1 \mid !A$
Monadic objects	M	$::= N \mid [N, M] \mid M_1 \otimes M_2 \mid 1 \mid !N$

As for typing these new constructs, first we consider the lax judgment and the rules pertaining to the monadic type constructor.

E is an expression of type S $\quad \Gamma; \Delta \vdash_\Sigma E \leftarrow S$

$$\frac{\Gamma; \Delta \vdash E \leftarrow S}{\Gamma; \Delta \vdash \{E\} \Leftarrow \{S\}}\ \{\}\mathbf{I}$$

$$\frac{\Gamma; \Delta_1 \vdash R \Rightarrow \{S_0\} \quad \Gamma; \Delta_2, p\hat{:}S_0 \vdash E \leftarrow S}{\Gamma; \Delta_1, \Delta_2 \vdash (\mathsf{let}\ \{p\} = R\ \mathsf{in}\ E) \leftarrow S}\ \{\}\mathbf{E}$$

Concurrent computations are represented by sequences of let bindings, each of which corresponds to an individual computation step yielding a result of synchronous type. Since all of the elimination rules for synchronous types are invertible (a defining property of synchronous connectives), we can establish canonical forms by requiring such eliminations to occur as early as possible, and in a definite order. This leads to a *pattern syntax* for synchronous types, and the elimination of such types becomes *pattern expansion*, as follows:

Patterns	p	$::= x \mid [u, p] \mid p_1 \otimes p_2 \mid 1 \mid !u$
Pattern contexts	Ψ	$::= \cdot \mid p\hat{:}S, \Psi$
Pattern expansion		$\Gamma; \Delta; \Psi \vdash_\Sigma E \leftarrow S$

The pattern expansion judgment decomposes the variables in each pattern p in Ψ into the appropriate contexts Γ or Δ:

$$\frac{\Gamma; \Delta; \Psi \vdash E \leftarrow S}{\Gamma; \Delta; 1\hat{:}1, \Psi \vdash E \leftarrow S}\ 1\mathbf{L} \qquad \frac{\Gamma, u\!:\!A; \Delta; \Psi \vdash E \leftarrow S}{\Gamma; \Delta; !u\hat{:}!A, \Psi \vdash E \leftarrow S}\ !\mathbf{L}$$

$$\frac{\Gamma, u\!:\!A; \Delta; p\hat{:}S_0, \Psi \vdash E \leftarrow S}{\Gamma; \Delta; [u, p]\hat{:}\exists x\!:\!A.\,S_0, \Psi \vdash E \leftarrow S}\ \exists\mathbf{L}$$

$$\frac{\Gamma; \Delta; p_1\hat{:}S_1, p_2\hat{:}S_2, \Psi \vdash E \leftarrow S}{\Gamma; \Delta; p_1 \otimes p_2\hat{:}S_1 \otimes S_2, \Psi \vdash E \leftarrow S}\ \otimes\mathbf{L}$$

$$\frac{\Gamma; \Delta, x\hat{:}A; \Psi \vdash E \leftarrow S}{\Gamma; \Delta; x\hat{:}A, \Psi \vdash E \leftarrow S}\ \mathrm{A}\mathbf{L} \qquad \frac{\Gamma; \Delta \vdash E \leftarrow S}{\Gamma; \Delta; \cdot \vdash E \leftarrow S}\ \leftarrow\leftarrow$$

At the *kernel* of each computation, following any let bindings representing the computation steps, there is a final result of type S. The terms representing such results are called *monadic objects*. The introduction rules for these objects lie in a new non-lax judgment:

M is a monadic object of type S $\quad \Gamma; \Delta \vdash_\Sigma M \Leftarrow S$

$$\frac{}{\Gamma; \cdot \vdash 1 \Leftarrow 1}\ 1\mathbf{I} \qquad \frac{\Gamma; \cdot \vdash N \Leftarrow A}{\Gamma; \cdot \vdash !N \Leftarrow !A}\ !\mathbf{I}$$

$$\frac{\Gamma; \cdot \vdash N \Leftarrow A \quad \Gamma; \Delta \vdash M \Leftarrow \mathsf{inst\_s}_A(u.\,S, N)}{\Gamma; \Delta \vdash [N, M] \Leftarrow \exists u\!:\!A.\,S}\ \exists\mathbf{I}$$

$$\frac{\Gamma; \Delta_1 \vdash M_1 \Leftarrow S_1 \quad \Gamma; \Delta_2 \vdash M_2 \Leftarrow S_2}{\Gamma; \Delta_1, \Delta_2 \vdash M_1 \otimes M_2 \Leftarrow S_1 \otimes S_2}\ \otimes\mathbf{I}$$

$$\frac{\Gamma; \Delta \vdash N \Leftarrow A}{\Gamma; \Delta \vdash N \leftarrow A}\ \Leftarrow\leftarrow$$

Let us recall the notation $\mathsf{inst\_a}_A(u.\,B, N)$ of Section 2 for the canonical type resulting from a substitution of a canonical object N into a canonical type B. For $\lambda^{\Pi\multimap\&\top}$ this definition is analogous to the ordinary notion of β-reduction. Having extended the type theory with the monadic type, we must extend the definition of inst_a as well. There is not room to present the inductive definition here;[5] however, the result will be as though the objects were substituted in the ordinary way, and then the reductions

$$\begin{aligned} &(\mathsf{let}\ \{p_1\} = \{\mathsf{let}\ \{p_2\} = R\ \mathsf{in}\ E_1\}\ \mathsf{in}\ E_2) \\ &\quad \mapsto (\mathsf{let}\ \{p_2\} = R\ \mathsf{in}\ \mathsf{let}\ \{p_1\} = \{E_1\}\ \mathsf{in}\ E_2)\,; \\ &\quad \mathsf{let}\ \{p\} = \{M\}\ \mathsf{in}\ E \mapsto [M/p]E\,, \end{aligned}$$

together with β-reductions, applied as often as necessary to reach a canonical form.

To return to the original question, has conservativity over $\lambda^{\Pi\multimap\&\top}$ been retained? Specifically, does the strong canonical forms property still hold for $\lambda^{\Pi\multimap\&\top\{\exists\otimes 1!\}}$? Yes, because the elimination rules for $\exists$, $\otimes$, 1 and ! are restricted to the lax judgment.

4.1 Representing dynamic semantics

Continuing with our running example concerning the π-calculus, we now present the operational semantics. Each state of the operational semantics is a pair $(\Theta; \Gamma)$ where Θ records the names that have been generated and Γ contains all the processes which are currently executing. An executing process is available at a given *multiplicity*, which we write 1 or ω, indicating that a process is available once, or arbitrarily many times, respectively.

$$\Gamma \quad ::= \cdot \mid \Gamma, P^1 \mid \Gamma, P^\omega$$

In order to streamline the presentation of the transition rules, we rely on the meta-notation $\Gamma \oplus P$. This denotes states of the form Γ', P^1, Γ'' where $\Gamma = \Gamma', \Gamma''$, or of the form $\Gamma', P^\omega, \Gamma''$ where $\Gamma = \Gamma', P^\omega, \Gamma''$. With this in mind, the transition rules can be written as follows.

$$\begin{aligned} (\Theta; \Gamma \oplus \mathsf{stop}) &\longrightarrow (\Theta; \Gamma) \\ (\Theta; \Gamma \oplus (P \mid Q)) &\longrightarrow (\Theta; \Gamma, P^1, Q^1) \\ (\Theta; \Gamma \oplus (\mathsf{repeat}\ P)) &\longrightarrow (\Theta; \Gamma, P^\omega) \\ (\Theta; \Gamma \oplus (\mathsf{new}(x\!:\!\tau); P)) &\longrightarrow (\Theta, x\!:\!\tau; \Gamma, P^1) \\ (\Theta; \Gamma \oplus (\mathsf{choose}\ P\ Q)) &\longrightarrow (\Theta; \Gamma, P^1) \\ (\Theta; \Gamma \oplus (\mathsf{choose}\ P\ Q)) &\longrightarrow (\Theta; \Gamma, Q^1) \\ (\Theta; \Gamma \oplus (\mathsf{out}\ x\langle y\rangle) \oplus (\mathsf{inp}\ x(z\!:\!\tau); P)) & \\ &\longrightarrow (\Theta; \Gamma, [y/z]P^1) \\ (\Theta; \Gamma \oplus (\mathsf{begin}\ L; P)) &\longrightarrow (\Theta; \Gamma, P^1) \\ (\Theta; \Gamma \oplus (\mathsf{end}\ L; P)) &\longrightarrow (\Theta; \Gamma, P^1) \end{aligned}$$

[5] See our technical report for exhaustive details [WCPW02].

The rule for new carries the proviso that x not be in the domain of Θ.

Computations $(\Theta, \Gamma) \longrightarrow (\Theta', \Gamma')$ are represented by CLF expressions

$$x_1 : \mathsf{nm}, \ldots, x_n : \mathsf{nm},\ r_1 : \mathsf{run}\ P_1, \ldots, r_i : \mathsf{run}\ P_i; \\ r_{i+1} \hat{:} \mathsf{run}\ P_{i+1}, \ldots, r_n \hat{:} \mathsf{run}\ P_n \ \vdash\ E \leftarrow \top$$

in a context having unrestricted hypotheses of type nm for each generated name, unrestricted hypotheses $r_1 \ldots r_i$ of type $\mathsf{run}\ P$ for each process P^ω that is executing and available at multiplicity ω, and linear hypotheses $r_{i+1} \ldots r_n$ of type $\mathsf{run}\ P$ for each process P^1 that is available at multiplicity 1. Here $\mathsf{run} : \mathsf{pr} \rightarrow \mathsf{type}$ is a new type constructor. The computation overall is required to have the additive unit type $\top$, meaning that computation can stop at any time, with any leftover resources (linear hypotheses) consumed by $\langle\rangle$, the introduction form for $\top$.

Then the dynamic semantics of each of the "structural" process constructors stop, par, repeat, and new can be represented by a corresponding synchronous CLF connective:

$$\begin{aligned} \mathsf{ev\_stop} &: \mathsf{run\ stop} \multimap \{1\}. \\ \mathsf{ev\_par} &: \mathsf{run}\ (\mathsf{par}\ P\ Q) \multimap \{\mathsf{run}\ P \otimes \mathsf{run}\ Q\}. \\ \mathsf{ev\_repeat} &: \mathsf{run}\ (\mathsf{repeat}\ P) \multimap \{!\mathsf{run}\ P\}. \\ \mathsf{ev\_new} &: \mathsf{run}\ (\mathsf{new}\ \tau\ (\lambda u.\, P\ u)) \multimap \{\exists u : \mathsf{nm}.\, \mathsf{run}\ (P\ u)\}. \end{aligned}$$

The remaining constructors are interpreted according to their semantics:

$$\begin{aligned} \mathsf{ev\_choose}_1 &: \mathsf{run}\ (\mathsf{choose}\ P_1\ P_2) \multimap \{\mathsf{run}\ P_1\}. \\ \mathsf{ev\_choose}_2 &: \mathsf{run}\ (\mathsf{choose}\ P_1\ P_2) \multimap \{\mathsf{run}\ P_2\}. \\ \mathsf{ev\_sync} &: \mathsf{run}\ (\mathsf{out}\ X\ Y) \multimap \mathsf{run}\ (\mathsf{inp}\ X\ \tau\ (\lambda y.\, P\ y)) \\ &\quad \multimap \{\mathsf{run}\ (P\ Y)\}. \\ \mathsf{ev\_begin} &: \Pi L : \mathsf{label}.\, \mathsf{run}\ (\mathsf{begin}\ L\ P) \multimap \{\mathsf{run}\ P\}. \\ \mathsf{ev\_end} &: \Pi L : \mathsf{label}.\, \mathsf{run}\ (\mathsf{end}\ L\ P) \multimap \{\mathsf{run}\ P\}. \end{aligned}$$

We depart from the usual practice of leaving outermost Π quantifiers implicit for reasons that will become clear in Section 6.

One interesting feature of the CLF encoding is that many of the structural equivalences of presentations of the π-calculus based on a notion of structural congruence appear automatically (shallowly) as consequences of the principles of exchange, weakening (since $\top$ is present) and so on satisfied by CLF hypotheses.

5 Equality modeling concurrency

In Section 2 the definitional equality of λ^Π was given as α-equivalence. Since an invariant of our formulation of λ^Π is that all well-formed objects are $\beta\eta$-canonical, $\beta\eta$-equivalences need not be considered. The invariant extends to CLF's type theory $\lambda^{\Pi\multimap\&\top\{\exists\otimes 1!\}}$, and indeed, one can show that α-equivalence is a reasonable notion of equality on the larger type theory. However, supposing r has type $\mathsf{run}\ (\mathsf{par}\ (\mathsf{begin}\ L_1\ P_1)\ (\mathsf{begin}\ L_2\ P_2))$, the expressions

$$\begin{aligned} &\mathsf{let}\ \{r_1 \otimes r_2\} = \mathsf{ev\_par}\,\hat{}\,r\ \mathsf{in} \\ &\quad \mathsf{let}\ \{r_1'\} = \mathsf{ev\_begin}\ L_1\,\hat{}\,r_1\ \mathsf{in}\ \mathsf{let}\ \{r_2'\} = \mathsf{ev\_begin}\ L_2\,\hat{}\,r_2\ \mathsf{in}\ \langle\rangle \\ &\mathsf{let}\ \{r_1 \otimes r_2\} = \mathsf{ev\_par}\,\hat{}\,r\ \mathsf{in} \\ &\quad \mathsf{let}\ \{r_2'\} = \mathsf{ev\_begin}\ L_2\,\hat{}\,r_2\ \mathsf{in}\ \mathsf{let}\ \{r_1'\} = \mathsf{ev\_begin}\ L_1\,\hat{}\,r_1\ \mathsf{in}\ \langle\rangle \end{aligned}$$

of the operational semantics given in Section 4 would not then be α-equivalent.

In applications, it is more natural to think of the *abstract* structure of a concurrent computation, so that computations that differ only in the order of occurrence of independent events are identified. We want to consider equality modulo *concurrency equations* of the form

$$(\mathsf{let}\ \{p_1\} = R_1\ \mathsf{in}\ \mathsf{let}\ \{p_2\} = R_2\ \mathsf{in}\ E) \ =\ (\mathsf{let}\ \{p_2\} = \\ R_2\ \mathsf{in}\ \mathsf{let}\ \{p_1\} = R_1\ \mathsf{in}\ E)$$

provided that the bindings of variables free in R_1 or R_2 are not thereby altered (which amounts to the *independence* of the two computation steps).

There is a notion of equality satisfying all properly formed (with respect to variable binding) occurrences of the equation above, and admitting all the congruences one would expect, as well as the axioms of an equivalence relation.[6] We adopt this notion as the definitional equality of CLF. In particular, for full $\lambda^{\Pi\multimap\&\top\{\exists\otimes 1!\}}$ the rule $\Rightarrow\Leftarrow$ of Section 2 is considered to refer to this new equality judgment.

The concurrent equality of CLF comes into its fullest power when we consider types *indexed* by monadic expressions, which can be used to model judgments over monadic expressions modulo concurrent equality. As a first (simple) example, we can use CLF's concurrent equality to model the multiset of labels that constitutes an effect in the static semantics of Section 3. Each label multiset $[L_1, \ldots, L_n]$ will be represented by an expression $\{\mathsf{let}\ \{1\} = \mathsf{latent}\ x\ L_1\ \mathsf{in}\ \ldots\ \mathsf{let}\ \{1\} = \mathsf{latent}\ x\ L_n\ \mathsf{in}\ 1\}$. Here $\mathsf{latent} : \mathsf{token} \rightarrow \mathsf{label} \rightarrow \{1\}$ is a primitive computation step generating no results. The parameter x of type token is used to ensure that these "computation steps" do not appear in executions of the dynamic semantics (excect within types mentioned in such executions). The parameter is otherwise ignored. The equality on this representation then naturally models equality of multisets.

The implementation of assume and $\mathsf{consume}$ then goes as follows:

$$\begin{aligned} \mathsf{token} &: \mathsf{type}. \\ \mathsf{eff} &= \mathsf{token} \rightarrow \{1\} : \mathsf{type}. \\ \mathsf{latent} &: \mathsf{token} \rightarrow \mathsf{label} \rightarrow \{1\}. \\ \mathsf{consume} &: \mathsf{eff} \rightarrow \mathsf{type}. \\ \mathsf{assume} &: \mathsf{eff} \rightarrow \mathsf{pr} \rightarrow \mathsf{type}. \end{aligned}$$

$$\begin{aligned} \mathsf{con\_eps} &: \mathsf{consume}\ (\lambda x.\, \{1\}) \multimapinv \top. \\ \mathsf{con\_join} &: \mathsf{consume}\ (\lambda x.\, \{\mathsf{let}\ \{1\} = \mathsf{latent}\ x\ L\ \mathsf{in} \\ &\qquad \mathsf{let}\ \{1\} = E\ x\ \mathsf{in}\ 1\} \\ &\qquad \multimapinv \mathsf{effect}\ L \multimapinv \mathsf{consume}\ E. \end{aligned}$$

$$\begin{aligned} \mathsf{ass\_eps} &: \mathsf{assume}\ (\lambda x.\, \{1\})\ P \multimapinv \mathsf{good}\ P. \\ \mathsf{ass\_join} &: \mathsf{assume}\ (\lambda x.\, \{\mathsf{let}\ \{1\} = \mathsf{latent}\ x\ L\ \mathsf{in} \\ &\qquad \mathsf{let}\ \{1\} = E\ x\ \mathsf{in}\ 1\} \\ &\qquad \multimapinv (\mathsf{effect}\ L \multimap \mathsf{assume}\ E\ P). \end{aligned}$$

A deeper example is shown in the next section, in which monadic expressions indexing judgments are used to model concurrent computations, rather than mere syntax. There, concurrent computations as index objects are used to define the correspondence property for an execution, allowing the safety of a process to be characterized.

[6] See our technical report [WCPW02] for the decidably presented, syntax-directed definition of this equality.

$$
\begin{array}{rcl}
\mathsf{corr} & : & \{\top\} \to \mathsf{type}. \\
\\
\mathsf{corr\_finish} & : & \mathsf{corr}\ \{\top\}. \\
\mathsf{corr\_stop} & : & \mathsf{corr}\ \{\mathsf{let}\ \{1\} = \mathsf{ev\_stop}\,\hat{}\,R\ \mathsf{in}\ \mathsf{let}\ \{\_\} = E\ \mathsf{in}\ \langle\rangle\} \leftarrow \mathsf{corr}\ E. \\
\mathsf{corr\_par} & : & \mathsf{corr}\ \{\mathsf{let}\ \{r_1 \otimes r_2\} = \mathsf{ev\_par}\,\hat{}\,R\ \mathsf{in}\ \mathsf{let}\ \{\_\} = E\,\hat{}\,r_1\,\hat{}\,r_2\ \mathsf{in}\ \langle\rangle\} \leftarrow (\Pi r_1.\, \Pi r_2.\, \mathsf{corr}\ (E\,\hat{}\,r_1\,\hat{}\,r_2)). \\
\mathsf{corr\_repeat} & : & \mathsf{corr}\ \{\mathsf{let}\ \{!r\} = \mathsf{ev\_repeat}\,\hat{}\,R\ \mathsf{in}\ \mathsf{let}\ \{\_\} = E\ r\ \mathsf{in}\ \langle\rangle\} \leftarrow (\Pi r.\, \mathsf{corr}\ (E\ r)). \\
\mathsf{corr\_new} & : & \mathsf{corr}\ \{\mathsf{let}\ \{[x, r]\} = \mathsf{ev\_new}\,\hat{}\,R\ \mathsf{in}\ \mathsf{let}\ \{\_\} = E\ x\,\hat{}\,r\ \mathsf{in}\ \langle\rangle\} \leftarrow (\Pi x.\, \Pi r.\, \mathsf{corr}\ (E\ x\,\hat{}\,r)). \\
\mathsf{corr\_choose}_i & : & \mathsf{corr}\ \{\mathsf{let}\ \{r\} = \mathsf{ev\_choose}_i\,\hat{}\,R\ \mathsf{in}\ \mathsf{let}\ \{\_\} = E\,\hat{}\,r\ \mathsf{in}\ \langle\rangle\} \leftarrow (\Pi r.\, \mathsf{corr}\ (E\ r)). \\
\mathsf{corr\_sync} & : & \mathsf{corr}\ \{\mathsf{let}\ \{r\} = \mathsf{ev\_sync}\,\hat{}\,R_1\,\hat{}\,R_2\ \mathsf{in}\ \mathsf{let}\ \{\_\} = E\,\hat{}\,r\ \mathsf{in}\ \langle\rangle\} \leftarrow (\Pi r.\, \mathsf{corr}\ (E\,\hat{}\,r)). \\
\mathsf{corr\_begin} & : & \mathsf{corr}\ \{\mathsf{let}\ \{r\} = \mathsf{ev\_begin}\ L\,\hat{}\,R\ \mathsf{in}\ \mathsf{let}\ \{\_\} = E\,\hat{}\,r\ \mathsf{in}\ \langle\rangle\} \\
 & & \leftarrow (\Pi r.\, \mathsf{effect}\ L \multimap \mathsf{corr}\ (E\,\hat{}\,r)). \\
\mathsf{corr\_end} & : & \mathsf{corr}\ \{\mathsf{let}\ \{r\} = \mathsf{ev\_end}\ L\,\hat{}\,R\ \mathsf{in}\ \mathsf{let}\ \{\_\} = E\,\hat{}\,r\ \mathsf{in}\ \langle\rangle\} \\
 & & \circ\!\!- \mathsf{effect} L \leftarrow (\Pi r.\, \mathsf{corr}\ (E\,\hat{}\,r)).
\end{array}
$$

Figure 3: Rules for the weak correspondence property

6 Concurrent computations indexing a type

Recall that an execution of a process is said to satisfy the correspondence property if every execution of end L is preceded by a distinct matching execution of begin L for the same label L. This property appears to make explicit reference to the sequencing of operations (and would thus not appear to be stable under the reordering of computation steps, as given by CLF's concurrency equations). However, the property may be reduced to an equivalent one that does respect concurrency equations, in the following way.

If an execution violates the correspondence property, then there is a serialization in which some end L_0 occurs without a matching distinct begin L_0. (We choose the first such end L_0 in the temporal order determined by the serialization.) The occurrence of this unmatched end L_0 slices the serialization into two parts: the execution steps coming before the "bad" end L_0 step (and including it), and the ones coming after it. But since we define the operational semantics such that execution can stop at any time, the "before" slice is itself an execution. (In essence, a serialization of an execution can be truncated at any intermediate state, yielding a shorter execution.)

But this truncated execution (ending with the "bad" occurrence of end L_0) not only violates the correspondence property, it also violates a weaker requirement: namely, that the *number* of occurrences of begin L be greater than or equal to the *number* of occurrences of end L, for each label L. (Otherwise, we would have been able to find a matching begin L_0.)

This latter, weaker correspondence property makes no reference to the temporal order of computation steps, so it is a good candidate for formalization in CLF. We can then characterize *safe* processes as those for which all execution objects admitted by the operational semantics of Section 4 satisfy the weak correspondence property.

The weak correspondence property is represented in CLF by a type family corr indexed by computations (which, recall, have CLF type $\{\top\}$). The rules characterizing this judgment are shown in Figure 3. The basic idea is similar to that employed by the static semantics of Section 3. An execution containing begin L is checked by adding a linear hypothesis of effect L to the context, and checking the remainder of the execution. An execution containing end L is accepted only if there is a linear hypothesis effect L available, in which case it is consumed, and checking proceeds over the rest of the execution. The other computation steps have no effect on the current multiset of linear hypotheses.

The preceding description should make it clear that corr is an *adequate* representation of the weak correspondence property (the proof being a simple induction over computations). The safety of a process is then indirectly characterized as the ability to generate, for each execution E of the process admitted by the operational semantics, a proof of the weak correspondence property for that execution (an object of type corr E).

7 Meta-theory

This section sketches the meta-theory of the canonical formulation of CLF. Additional details may be found in our technical report [WCPW02].

7.1 Identity and substitution properties

As discussed in Section 2, the CLF framework syntactically restricts the form of objects so that they will always be canonical. This is a good design choice in the logical frameworks context, but it carries with it the obligation to ensure that the underlying logic (via the Curry-Howard isomorphism, if you like) is sensible. In particular, the principles of *identity* and *substitution* must hold.

> **Identity.** *Unrestricted case*: For any Γ and A, $\Gamma, u\!:\!A; \cdot \vdash N \Leftarrow A$ for some N. *Linear case*: For any Γ and A, $\Gamma; x\hat{:}A \vdash N \Leftarrow A$ for some N.

> **Substitution.** *Unrestricted case*: if $\Gamma; \cdot \vdash N_0 \Leftarrow A$ and $\Gamma, u\!:\!A; \Delta \vdash N \Leftarrow C$ then $\Gamma; \Delta \vdash N' \Leftarrow C'$ for some N', where C' is an appropriate substitution instance of C. *Linear case*: if $\Gamma; \Delta_1 \vdash N_0 \Leftarrow A$ and $\Gamma; \Delta_2, x\hat{:}A \vdash N \Leftarrow C$ then $\Gamma; \Delta_1, \Delta_2 \vdash N' \Leftarrow C$ for some N'.

In the standard reduction-oriented treatment of proofs, these are fairly trivial, because variables and general terms are in the same syntactic category. Substitution simply syntactically replaces the target variable with the substituend—possibly creating redices. Here, redices are not syntactically allowed, and variables are syntactically *atomic* while general terms are syntactically *normal*, so it is not possible to directly replace a variable with a substituend. By the same token, a variable of higher type cannot stand by itself as a canonical object—canonical objects of higher type must be introduction forms—so the identity principle cannot be witnessed by a bare variable.

Instead, the meta-theory of CLF relies on *algorithms* that *compute* witnesses to the identity and substitution principles. These are, respectively, the *expansion algorithm* and the *instantiation algorithm*.[7]

Principle	Substitution	Identity
Algorithm	Instantiation	Expansion
Supersedes	β-normalization	η-normalization
Notation	$\mathsf{inst\_n}_A(x.N, N_0) \equiv N$	$\mathsf{expand}_A(R) \equiv N$

Think of the instantiation operator $\mathsf{inst\_n}_A(x.N, N_0)$ as an algorithm for computing the canonical form of the result of instantiating the variable x in the object N with the object N_0. The instantiation operator is indexed by the type A of the substituend N_0. If A is a base type, we have $\mathsf{inst\_n}_A(x.N, N_0) = [N_0/x]N$; that is, instantiation reduces to ordinary syntactic substitution. At higher type more complex situations arise.

Dually, we think of the expansion operator $\mathsf{expand}_A(R)$ as computing the canonical form of the atomic object R of putative type A. This is analogous to η-expansion, except that the term R and its expansion inhabit different syntactic categories if A is a higher type.

These algorithms must be (and are) effectively presented, because the typing judgment of the full dependent type theory appeals to instantiation, and effective typing is central to the logical framework concept. The use of the instantiation algorithm in dependent typing has a further important ramification: the instantiation algorithm must be *effective on ill-typed terms*. Otherwise, there is a circularity between instantiation and typing, leading to a very complex meta-theory.[8] Since the substitution principle does not hold for ill-typed terms, we allow the witnessing instantiation algorithm to report failure or yield garbage on ill-typed input; e.g., $\mathsf{inst\_n}_A(x.x\ x, \lambda x.x\ x) \equiv \mathsf{fail}$. Garbage in, garbage out, but at least we get our garbage out in finite time!

7.2 Instantiation

Space constraints preclude the incorporation of all the cases of the definitions of these operators. Full details are available, of course, in our technical report [WCPW02].

We begin by examining the cases for the LF fragment of instantiation, shown in Figure 4. The recurrence defining instantiation is based on the observation, exploited in cut elimination proofs on the logical side [Pfe00], but not so well known on the type theoretic side, that the canonical result of substituting one canonical term into another can be defined by induction on the type of the term being substituted. Accordingly, the instantiation operators are defined as a family parameterized over the type of the object being substituted. In the notation $\mathsf{inst\_}c_A(x.X, N)$ this type A appears as a subscript. Here c is replaced by a mnemonic for the particular syntactic category to which the instantiation operator applies. The variable x is to be considered bound within the term X (of whatever category) being substituted into. The operators defined in this section should be thought of as applying to equivalence classes of concrete terms modulo α-equivalence on bound variables.

Together with the instantiation operators, and defined by mutual recursion with them, is a *reduction operator* $\mathsf{reduce}_A(x.R, N)$ that computes the canonical object resulting from the instantiation of x with N in the case that the *head variable* head(R) of the atomic object R is x. Thus, roughly speaking, it corresponds to the idea of weak head reduction for systems with β-reduction. The instantiation operator $\mathsf{inst\_r}_A(x.R, N)$, by contrast, is only defined if the head of R is *not* x. Another distinguishing feature is that reduction on an atomic object yields a normal object, while instantiation on an atomic object yields an atomic object.

Finally, there is a *type reduction operator* $\mathsf{treduce}_A(x.R)$ that computes the putative type of R given that the head of R is x and the type of x is A. [9] Type reduction is used in side conditions that ensure that the recurrence defining instantiation is well-founded.

The recurrence defining these operators is based on a structural induction. There is an outer induction on the type subscripting the operators, and an inner simultaneous induction on the two arguments. Noting first that if $\mathsf{treduce}_A(x.R)$ is defined, it is a subterm of A, the fact that the recurrence relations respect this induction order can be verified almost by inspection. The only slightly subtle case is the equation for $\mathsf{reduce}_A(x.R\ N, N_0)$, which is the only case in which the subscripting type changes. Here the side condition $\mathsf{treduce}_A(x.R) \equiv \Pi x{:}B.C$ ensures that B must be a strict subterm of A for the reduction to be defined. An instantiation such as $\mathsf{inst\_n}_A(x.x\ x, \lambda x.x\ x)$ is guaranteed to fail the side condition after only finitely many expansions of the recurrence.

Another way in which an instance of the instantiation operators might fail to be defined would be if the recursive instantiation $\mathsf{inst\_r}_A(x.R, N_0)$ in the same equation failed to result in a manifest lambda abstraction $\lambda y.N'$. In fact, this could only happen if the term N_0 failed to have the ascribed type A. [10] So instantiation always terminates, regardless of whether its arguments are well typed, but it is not defined in all cases. After the meta-theory is further developed, it can be shown that instantiation is always defined on well-typed terms when the types match in the appropriate way.

The cases of instantiation involving the monad, shown in Figure 5, are not without interest. These lean heavily on prior work on proof term assignments for modal logics [PD01].

In order to extend instantiation to the full CLF language, with its pattern-oriented destructor for the monadic type, it is necessary to introduce *matching operators* $\mathsf{match\_}c_S(p.E, X)$, where X is either an expression or a monadic object. The matching operator computes the result of instantiating E according to the substitution on the variables of p generated by matching p against X. (The variables in p should be considered bound in E.) In the case that X is a monadic object M_0, this is straightforward: the syntax of monadic objects corresponds precisely to that of patterns. But in the case that X is a let binding, an interesting issue arises:

$$\mathsf{match\_e}_S(p.\,\mathsf{let}\ \{p_1\} = R_1\ \mathsf{in}\ E_1, \mathsf{let}\ \{p_2\} = R_2\ \mathsf{in}\ E_2) \equiv ?$$

The key is found in Pfenning and Davies' non-standard substitutions for the proof terms of the modal logics of possibility and laxity [PD01]. These analyze the structure of the

[7] Here and in the reminder we use x generically for either a linear or unrestricted variable.

[8] This circularity, which the present treatment of CLF avoids, is analogous to the difficulties encountered in the early reduction-oriented treatments of LF, where typing refers to equality, which is decided by normalization, but normalization is only effective for well-typed terms.

[9] Actually, to be more precise, the type of R will be a substitution instance of $\mathsf{treduce}_A(x.R)$. The instantiation operators do not keep track of dependencies within the type subscript.

[10] Or a substitution instance of A.

$\mathsf{treduce}_A(x.\,R) \equiv B$ [Type reduction]

$$\mathsf{treduce}_A(x.\,x) \equiv A$$
$$\mathsf{treduce}_A(x.\,R\ N) \equiv C \quad \text{if } \mathsf{treduce}_A(x.\,R) \equiv \Pi x{:}B.\,C$$

$\mathsf{reduce}_A(x.\,R, N_0) \equiv N'$ [Reduction]

$$\mathsf{reduce}_A(x.\,x, N_0) \equiv N_0$$
$$\mathsf{reduce}_A(x.\,R\ N, N_0) \equiv \mathsf{inst\_n}_B(y.\,N', \mathsf{inst\_n}_A(x.\,N, N_0))$$
$$\text{if } \mathsf{treduce}_A(x.\,R) \equiv \Pi x{:}B.\,C \text{ and } \mathsf{reduce}_A(x.\,R, N_0) \equiv \lambda y.\,N'$$

$\mathsf{inst\_r}_A(x.\,R, N_0) \equiv R'$ [Atomic object instantiation]

$$\mathsf{inst\_r}_A(x.\,c, N_0) \equiv c$$
$$\mathsf{inst\_r}_A(x.\,y, N_0) \equiv y \quad \text{if } y \text{ is not } x$$
$$\mathsf{inst\_r}_A(x.\,R\ N, N_0) \equiv (\mathsf{inst\_r}_A(x.\,R, N_0))\ (\mathsf{inst\_n}_A(x.\,N, N_0))$$

$\mathsf{inst\_n}_A(x.\,N, N_0) \equiv N'$ [Normal object instantiation]

$$\mathsf{inst\_n}_A(x.\,\lambda y.\,N, N_0) \equiv \lambda y.\,\mathsf{inst\_n}_A(x.\,N, N_0) \quad \text{if } y \notin \mathrm{FV}(N_0)$$
$$\mathsf{inst\_n}_A(x.\,R, N_0) \equiv \mathsf{inst\_r}_A(x.\,R, N_0) \quad \text{if head}(R) \text{ is not } x$$
$$\mathsf{inst\_n}_A(x.\,R, N_0) \equiv \mathsf{reduce}_A(x.\,R, N_0) \quad \text{if } \mathsf{treduce}_A(x.\,R) \equiv a$$

Figure 4: Instantiation, LF fragment

object being substituted, not, as in the usual case, the term being substituted into. The effect is similar to a commuting conversion:

$$\mathsf{match\_e}_S(p.\,\mathsf{let}\ \{p_1\} = R_1\ \mathsf{in}\ E_1, \mathsf{let}\ \{p_2\} = R_2\ \mathsf{in}\ E_2) \equiv$$
$$(\mathsf{let}\ \{p_2\} = R_2\ \mathsf{in}\ \mathsf{match\_e}_S(p.\,\mathsf{let}\ \{p_1\} = R_1\ \mathsf{in}\ E_1, E_2))$$

It is interesting that both non-standard substitution and pattern matching—the latter not present in Pfenning and Davies' system—rely in this way on an analysis of the object being substituted rather than the term being substituted into. In a sense, this commonality is what makes the harmonious interaction between CLF's modality and its synchronous types possible.

The induction order mentioned above leads immediately to the following theorem.

Theorem 1 (Definability of instantiation) *The recurrence for the reduction, instantiation, and matching operators uniquely determines the least partial functions (up to α-equivalence) solving them.*

Proof: The proof is by an outer structural induction on the type subscript, and an inner simultaneous structural induction on the two arguments. ■

7.3 Expansion

The definition of expansion is shown in Figure 6. In some cases, new bound variables are introduced on the right-hand side of an equation. Any new variables in an instance of such an equation are required to be distinct from one another and from any other variables in the equation instance.

Again there is a definability theorem based on the induction order implicit in the equations.

Theorem 2 (Definability of expansion) *1. If $\mathsf{pexpand}_S(p_1)$ and $\mathsf{pexpand}_S(p_2)$ are both defined then p_1 and p_2 are the same up to variable renaming.*

2. Given S, there is a pattern p, fresh with respect to any given set of variables, such that $\mathsf{pexpand}_S(p)$ is defined.

3. The recurrence for expansion uniquely determines it as a total function up to α-equivalence.

Proof: The first part is by induction on S. The second and third parts are by induction on the type subscript, using the first part to ensure that the result of $\mathsf{expand}_{\{S\}}(R)$ is unique up to α-equivalence. ■

7.4 Further results

The following proposition is proved in the technical report [WCPW02]. The identity and substitution principles follow immediately.

Proposition 3 (Identity and substitution principles) *The following rules are admissible.*

$$\frac{\Gamma; \Delta \vdash R \Rightarrow A}{\Gamma; \Delta \vdash \mathsf{expand}_A(R) \Leftarrow A}$$

$$\frac{\Gamma; \cdot \vdash N_0 \Leftarrow A \quad \Gamma, x{:}A; \Delta \vdash N \Leftarrow C}{\Gamma; \Delta \vdash \mathsf{inst\_n}_A(x.\,N, N_0) \Leftarrow \mathsf{inst\_a}_A(x.\,N, C)}$$

$$\frac{\Gamma; \Delta_1 \vdash N_0 \Leftarrow A \quad \Gamma; \Delta_2, x \hat{:} A \vdash N \Leftarrow C}{\Gamma; \Delta_1, \Delta_2 \vdash \mathsf{inst\_n}_A(x.\,N, N_0) \Leftarrow C}$$

Lemmas concerning the algebraic laws satisfied by expansion and instantiation (roughly analogous to confluence

$\mathsf{inst\_n}_A(x.\,N, N_0) \equiv N'$ [Normal object instantiation, extended]

$$\mathsf{inst\_n}_A(x.\,\{E\}, N_0) \equiv \{\mathsf{inst\_e}_A(x.\,E, N_0)\}$$

$\mathsf{inst\_m}_A(x.\,M, N_0) \equiv M'$ [Monadic object instantiation]

$$\begin{aligned}
&\mathsf{inst_m}_A(x.\,M_1 \otimes M_2, N_0) \equiv \mathsf{inst_m}_A(x.\,M_1, N_0) \otimes \mathsf{inst_m}_A(x.\,M_2, N_0)\\
&\mathsf{inst_m}_A(x.\,1, N_0) \equiv 1\\
&\mathsf{inst_m}_A(x.\,!N, N_0) \equiv !(\mathsf{inst_n}_A(x.\,N, N_0))\\
&\mathsf{inst_m}_A(x.\,[N, M], N_0) \equiv [\mathsf{inst_n}_A(x.\,N, N_0), \mathsf{inst_m}_A(x.\,M, N_0)]\\
&\mathsf{inst_m}_A(x.\,N, N_0) \equiv \mathsf{inst_n}_A(x.\,N, N_0)
\end{aligned}$$

$\mathsf{inst\_e}_A(x.\,E, N_0) \equiv E'$ [Expression instantiation]

$$\begin{aligned}
&\mathsf{inst_e}_A(x.\,\mathsf{let}\ \{p\} = R\ \mathsf{in}\ E, N_0) \equiv (\mathsf{let}\ \{p\} = \mathsf{inst_r}_A(x.\,R, N_0)\ \mathsf{in}\ \mathsf{inst_e}_A(x.\,E, N_0))\\
&\qquad \text{if head}(R) \text{ is not } x,\\
&\qquad \text{and } \mathrm{FV}(p) \cap \mathrm{FV}(N_0) \text{ is empty}\\
&\mathsf{inst_e}_A(x.\,\mathsf{let}\ \{p\} = R\ \mathsf{in}\ E, N_0) \equiv \mathsf{match_e}_S(p.\,\mathsf{inst_e}_A(x.\,E, N_0), E')\\
&\qquad \text{if } \mathsf{treduce}_A(x.\,R) \equiv \{S\},\ \mathsf{reduce}_A(x.\,R, N_0) \equiv \{E'\},\\
&\qquad \text{and } \mathrm{FV}(p) \cap \mathrm{FV}(N_0) \text{ is empty}\\
&\mathsf{inst_e}_A(x.\,M, N_0) \equiv \mathsf{inst_m}_A(x.\,M, N_0)
\end{aligned}$$

$\mathsf{match\_m}_S(p.\,E, M_0) \equiv E'$ [Match monadic object]

$$\begin{aligned}
&\mathsf{match_m}_{S_1 \otimes S_2}(p_1 \otimes p_2.\,E, M_1 \otimes M_2) \equiv \mathsf{match_m}_{S_2}(p_2.\,\mathsf{match_m}_{S_1}(p_1.\,E, M_1), M_2)\\
&\qquad \text{if } \mathrm{FV}(p_2) \cap \mathrm{FV}(M_1) \text{ is empty}\\
&\mathsf{match_m}_1(1.\,E, 1) \equiv E\\
&\mathsf{match_m}_{!A}(!x.\,E, !N) \equiv \mathsf{inst_e}_A(x.\,E, N)\\
&\mathsf{match_m}_{\exists x\,:\,A.\,S}([x, p].\,E, [N, M]) \equiv \mathsf{match_m}_S(p.\,\mathsf{inst_e}_A(x.\,E, N), M)\\
&\mathsf{match_m}_A(x.\,E, N) \equiv \mathsf{inst_e}_A(x.\,E, N)
\end{aligned}$$

$\mathsf{match\_e}_S(p.\,E, E_0) \equiv E'$ [Match expression]

$$\begin{aligned}
&\mathsf{match_e}_S(p.\,E, \mathsf{let}\ \{p_0\} = R_0\ \mathsf{in}\ E_0) \equiv \mathsf{let}\ \{p_0\} = R_0\ \mathsf{in}\ \mathsf{match_e}_S(p.\,E, E_0)\\
&\qquad \text{if } \mathrm{FV}(p_0) \cap \mathrm{FV}(E) \text{ and } \mathrm{FV}(p) \cap \mathrm{FV}(E_0) \text{ are empty}\\
&\mathsf{match_e}_S(p.\,E, M_0) \equiv \mathsf{match_m}_S(p.\,E, M_0)
\end{aligned}$$

Figure 5: Instantiation, extended

$\mathsf{expand}_A(R) \equiv N$ [Expansion]

$$
\begin{aligned}
&\mathsf{expand}_a(R) \equiv R\\
&\mathsf{expand}_{A \multimap B}(R) \equiv \hat{\lambda} x.\, \mathsf{expand}_B(R\,\hat{}\,(\mathsf{expand}_A(x))) \quad \text{if } x \notin \mathrm{FV}(R)\\
&\mathsf{expand}_{\Pi x : A.B}(R) \equiv \lambda x.\, \mathsf{expand}_B(R\ (\mathsf{expand}_A(x))) \quad \text{if } x \notin \mathrm{FV}(R)\\
&\mathsf{expand}_{A \& B}(R) \equiv \langle \mathsf{expand}_A(\pi_1 R), \mathsf{expand}_B(\pi_2 R)\rangle\\
&\mathsf{expand}_{\top}(R) \equiv \langle\rangle\\
&\mathsf{expand}_{\{S\}}(R) \equiv (\mathsf{let}\ \{p\} = R\ \mathsf{in}\ \mathsf{pexpand}_S(p))
\end{aligned}
$$

$\mathsf{pexpand}_S(p) \equiv M$ [Pattern expansion]

$$
\begin{aligned}
&\mathsf{pexpand}_{S_1 \otimes S_2}(p_1 \otimes p_2) \equiv \mathsf{pexpand}_{S_1}(p_1) \otimes \mathsf{pexpand}_{S_2}(p_2)\\
&\mathsf{pexpand}_1(1) \equiv 1\\
&\mathsf{pexpand}_{!A}(!x) \equiv !(\mathsf{expand}_A(x))\\
&\mathsf{pexpand}_{\exists x : A.S}([x, p]) \equiv [\mathsf{expand}_A(x), \mathsf{pexpand}_S(p)]\\
&\mathsf{pexpand}_A(x) \equiv \mathsf{expand}_A(x)
\end{aligned}
$$

Figure 6: Expansion

results) and concerning the interaction of equality and instantiation are required. Other notable theorems (some of which are required in order to prove the result above) include the following.

Theorem 4 (Decidability of instantiation and expansion) *It is decidable whether any instance of the instantiation and expansion operators is defined, and if so, it can be effectively computed.*

Proof: For instantiation, this is proved by a simultaneous structural induction on the substituend, the term substituted into, and the putative type of the substituend. For expansion, the induction is over the structure of the type. ■

The inference rules for typing are structured in a syntax-directed manner, leading to a very simple proof of decidability. This is a substantial technical improvement over prior presentations of even the LF sublanguage alone.

Theorem 5 (Decidability of typing) *It is decidable whether any instance of the typing judgments is derivable.*

Proof: By structural induction on the subject of the judgments. This result requires the decidability of concurrent equality, for which see [WCPW02]. ■

The interaction of equality and substitution is particularly important, since CLF's equality is where concurrency enters. Thus, the following theorems describe, in essence, how concurrent computations modeled in our framework compose.

Theorem 6 *Concurrent equality $N_1 = N_2$ is an equivalence relation.*

Proof: Reflexivity, symmetry, and transitivity can each be proved by structural inductions (with appropriate lemmas, also proved by structural induction) [WCPW02]. ■

Theorem 7 *If $N = N'$ and $N_0 = N_0'$ then*

$$\mathsf{inst\_n}_A(x.\,N, N_0) = \mathsf{inst\_n}_A(x.\,N', N_0'),$$

assuming one side or the other is defined.

Proof: The proof appeals to composition laws for instantiation and a number of other technical lemmas. The inductive proofs of these lemmas and the main theorem follow the same induction order as for the decidability result [WCPW02]. ■

Theorem 8 *If $R = R'$ then* $\mathsf{expand}_A(R) = \mathsf{expand}_A(R')$.

Proof: This follows by structural induction on A. ■

8 Related work

Right from its inception, linear logic [Gir87] has been advocated as a logic with an intrinsic notion of state and concurrency. In the literature, many connections between concurrent calculi and linear logic have been observed. Due to space constraints we cannot survey this relatively large literature here. In a logical framework, we remove ourselves by one degree from the actual semantics; we represent rather than embed calculi. Thereby, CLF provides another point of view on many of the prior investigations.

Most closely related to our work is Miller's logical framework Forum [Mil94], which is based on a sequent calculus for classical linear logic and focusing proofs [And92]. As shown by Miller and elaborated by Chirimar [Chi95], Forum can also represent concurrency. Our work extends Forum in several directions. Most importantly, it is a type theory based on natural deduction and therefore offers an internal notion of proof object that is not available in Forum. Among other things, this means we can explicitly represent relations on deductions and therefore on concurrent computations.

There have been several formalizations of versions of the π-calculus in a variety of reasoning systems, such as HOL [Mel95], Coq [Hir97, HMS01], Isabelle/HOL [RHB01] or Linc [MT03, Tiu04]. A distinguishing feature of our sample encoding in this paper is the simultaneous use of higher-order abstract syntax, linearity, modality, and the intrinsic notion of concurrent computations which is exploited in the specification of the safety property. Also, we are not aware of a formal treatment of correspondence assertions or dependent effect typing for the π-calculus. On the other hand, at present we cannot formally reason *about* properties of our encodings yet, which is achieved to varying degrees in the case studies above.

Systems based on rewriting logic, such as Maude [Mes02], also natively support concurrent specifications (and have been used to model Petri nets, CCS, the π-calculus, etc). Moreover, Maude is able to model true concurrency of an object language via the concurrency that is intrinsic in rewriting of non-overlapping or properly nested independent redexes. One clear difference between systems is that Maude lacks operators comparable to CLF's dependent types and therefore intrinsic notions of substitution.

As already mentioned above, CLF is a conservative extension of LLF with the asynchronous connectives $\otimes$, 1, $!$, and $\exists$, encapsulated in a monad. The idea of monadic encapsulation goes back to Moggi's monadic metalanguage [Mog91] and is used heavily in functional programming. Our formulation follows the judgmental presentation of Pfenning and Davies [PD01] that completely avoids the need for commuting conversions, but treats neither linearity nor the existence of normal forms. This permits us to reintroduce some equations to model true concurrency in a completely orthogonal fashion.

9 Conclusions

The goal of this work has been to extend the elegant and logically motivated representation strategies for syntax, judgments, and state available in LF and LLF to the concurrent world. We have shown how the availability of a *notation* for concurrent executions, admitting a proper truly concurrent equality, enables powerful strategies for specifying properties of such executions. A crucial element is the use of the monadic type, ensuring that the extension to CLF is conservative in a strong sense.

Ultimately, it should become as simple and natural to manipulate the objects representing concurrent executions as it is to manipulate LF objects. If higher-order abstract syntax means never having to code up α-conversion or capture-avoiding substitution ever again, we hope that in the same way, the techniques explored here can make it unnecessary to code up multiset equality or concurrent equality ever again, so that intellectual effort can be focused on reasoning about deeper properties of concurrent systems.

In order to realize this, it will be necessary to extend CLF in all the many ways that LF has been fleshed out over the years by many researchers. There should be an implementation of the natural logic programming semantics for concurrent signatures, including non-deterministic, exploratory executions as well as exhaustive search of the state space. A decidable and acceptably efficient notion of unification (possibly with constraints, as in Twelf [PS99]) must be identified. We need strategies for writing down meta-theorems formally and checking them mechanically; for instance, we should be able to prove a soundness result for the π-calculus static semantics of Section 3. The success of analogous developments for LF leading to the Twelf implementation is a hopeful sign, but much work remains.

References

[And92] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):197–347, 1992.

[Chi95] Jawahar Lal Chirimar. *Proof Theoretic Approach to Specification Languages*. PhD thesis, University of Pennsylvania, May 1995.

[CP02] Iliano Cervesato and Frank Pfenning. A linear logical framework. *Information & Computation*, 179(1):19–75, November 2002.

[CPWW02] Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A concurrent logical framework II: Examples and applications. Technical Report CMU-CS-02-102, Department of Computer Science, Carnegie Mellon University, 2002. Revised May 2003.

[Gir87] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[GJ03] Andrew D. Gordon and Alan Jeffrey. Typing correspondence assertions for communication protocols. *Theoretical Computer Science*, 300(1–3):379–409, May 2003.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[Hir97] Daniel Hirschkoff. A full formalisation of pi-calculus theory in the Calculus of Constructions. In E. Gunter and A.P. Felty, editors, *Proceedings of the 10th International Conference on Theorem Proving in Higher-Order Logics (TPHOLs'97)*, pages 153–169, Murray Hill, New Jersey, USA, August 1997. Springer Verlag LNCS 1275.

[HMS01] Furio Honsell, Marino Miculan, and Ivan Scagnetto. Pi-calculus in (co)inductive type theories. *Theoretical Computer Science*, 253(2):239–285, 2001.

[IP98] Samin Ishtiaq and David Pym. A relevant analysis of natural deduction. *Journal of Logic and Computation*, 8(6):809–838, 1998.

[Mel95] Tom Melham. A mechanized theory of the pi-calculus in HOL. *Nordic Journal of Computing*, 1(1):50–76, 1995.

[Mes02] José Meseguer. Software specification and verification in rewriting logic. Lecture notes for the Marktoberdorf International Summer School, Germany, August 2002.

[Mil94] Dale Miller. A multiple-conclusion meta-logic. In S. Abramsky, editor, *Ninth Annual Symposium on Logic in Computer Science*, pages 272–281, Paris, France, July 1994. IEEE Computer Society Press.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[MT03] Dale Miller and Alwen Tiu. A proof theory for generic judgments. In P. Kolaitis, editor, *Proceedings of the 18th Annual Symposium on Logic in Computer Science (LICS'03)*, pages 118–127, Ottawa, Canada, June 2003. IEEE Computer Society Press.

[PD01] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11:511–540, 2001. Notes to an invited talk at the *Workshop on Intuitionistic Modal Logics and Applications* (IMLA'99), Trento, Italy, July 1999.

[Pfe00] Frank Pfenning. Structural cut elimination I. intuitionistic and classical logic. *Information and Computation*, 157(1/2):84–141, March 2000.

[PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.

[Rep99] John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[RHB01] Christine Röckl, Daniel Hirschkoff, and Stefan Berghofer. Higher-order abstract syntax with induction in Isabelle/HOL: Formalizing the pi-calculus and mechanizing the theory of contexts. In F. Honsell and M. Miculan, editors, *Proceedings of the 4th International Conference on Foundations of Software Science and Computation Structures (FOSSACS'01)*, pages 364–378, Genova, Italy, April 2001. Springer Verlag LNCS 2030.

[Tiu04] Alwen Tiu. *A Logical Framework for Reasoning about Logical Specifications*. PhD thesis, Pennsylvania State University, May 2004.

[WCPW02] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University, 2002. Revised May 2003.

[WCPW04] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework: The propositional fragment. In *Types for Proofs and Programs*. Springer-Verlag LNCS, 2004. Selected papers from the *Third International Workshop* Torino, Italy, April 2003. To appear.

[WL93] Thomas Y.C. Woo and Simon S. Lam. A semantic model for authentication protocols. In *IEEE Computer Society Symposium on Research in Security and Privacy*, pages 178–194, Oakland, CA, May 1993.

A CLF type theory summarized

See the technical report [WCPW02] for further details.

Syntax.

$$
\begin{array}{rcl}
K, L & ::= & \mathsf{type} \mid \Pi u\!:\!A.\, K \\
A, B, C & ::= & A \multimap B \mid \Pi u\!:\!A.\, B \mid A \,\&\, B \\
 & & \mid \top \mid \{S\} \mid P \\
P & ::= & a \mid P\; N \\
S & ::= & \exists u\!:\!A.\, S \mid S_1 \otimes S_2 \mid 1 \mid {!A} \mid A \\
\Gamma & ::= & \cdot \mid \Gamma, u\!:\!A \\
\Delta & ::= & \cdot \mid \Delta, x\hat{:}A \\
\Sigma & ::= & \cdot \mid \Sigma, a\!:\!K \mid \Sigma, c\!:\!A
\end{array}
$$

$$
\begin{array}{rcl}
N & ::= & \hat{\lambda} x.\, N \mid \lambda u.\, N \mid \langle N_1, N_2 \rangle \\
 & & \mid \langle\rangle \mid \{E\} \mid R \\
R & ::= & c \mid u \mid x \mid R\,\hat{}\,N \mid R\; N \mid \pi_1 R \mid \pi_2 R \\
E & ::= & \mathsf{let}\ \{p\} = R\ \mathsf{in}\ E \mid M \\
M & ::= & [N, M] \mid M_1 \otimes M_2 \mid 1 \mid {!N} \mid N \\
p & ::= & [u, p] \mid p_1 \otimes p_2 \mid 1 \mid {!u} \mid x \\
\Psi & ::= & p\hat{:}S, \Psi \mid \cdot
\end{array}
$$

Typing.

$$
\begin{array}{c}
\Gamma \vdash_\Sigma K \Leftarrow \mathsf{kind} \\
\Gamma \vdash_\Sigma A \Leftarrow \mathsf{type} \\
\Gamma \vdash_\Sigma P \Rightarrow K \\
\Gamma \vdash_\Sigma S \Leftarrow \mathsf{type}
\end{array}
\qquad
\begin{array}{c}
\Gamma; \Delta \vdash_\Sigma N \Leftarrow A \\
\Gamma; \Delta \vdash_\Sigma R \Rightarrow A \\
\Gamma; \Delta \vdash_\Sigma E \leftarrow S \\
\Gamma; \Delta; \Psi \vdash_\Sigma E \leftarrow S \\
\Gamma; \Delta \vdash_\Sigma M \Leftarrow S
\end{array}
\qquad
\begin{array}{c}
\vdash \Sigma\ \mathsf{ok} \\
\vdash_\Sigma \Gamma\ \mathsf{ok} \\
\Gamma \vdash_\Sigma \Delta\ \mathsf{ok} \\
\Gamma \vdash_\Sigma \Psi\ \mathsf{ok}
\end{array}
$$

$$
\begin{array}{c}
\mathsf{inst\_k}_A(u.\, K, N) = K' \\
\mathsf{inst\_a}_A(u.\, B, N) = B' \\
\mathsf{inst\_s}_A(u.\, S, N) = S'
\end{array}
$$

$$
\frac{}{\vdash \cdot\ \mathsf{ok}}
\qquad
\frac{\vdash \Sigma\ \mathsf{ok} \quad \cdot \vdash_\Sigma K \Leftarrow \mathsf{kind}}{\vdash \Sigma, a\!:\!K\ \mathsf{ok}}
\qquad
\frac{\vdash \Sigma\ \mathsf{ok} \quad \cdot \vdash_\Sigma A \Leftarrow \mathsf{type}}{\vdash \Sigma, c\!:\!A\ \mathsf{ok}}
$$

$$
\frac{}{\vdash_\Sigma \cdot\ \mathsf{ok}}
\qquad
\frac{\vdash_\Sigma \Gamma\ \mathsf{ok} \quad \Gamma \vdash_\Sigma A \Leftarrow \mathsf{type}}{\vdash_\Sigma \Gamma, u\!:\!A\ \mathsf{ok}}
$$

$$
\frac{}{\Gamma \vdash_\Sigma \cdot\ \mathsf{ok}}
\qquad
\frac{\Gamma \vdash_\Sigma \Delta\ \mathsf{ok} \quad \Gamma \vdash_\Sigma A \Leftarrow \mathsf{type}}{\Gamma \vdash_\Sigma \Delta, x\hat{:}A\ \mathsf{ok}}
$$

$$
\frac{}{\Gamma \vdash_\Sigma \cdot\ \mathsf{ok}}
\qquad
\frac{\Gamma \vdash_\Sigma S \Leftarrow \mathsf{type} \quad \Gamma \vdash_\Sigma \Psi\ \mathsf{ok}}{\Gamma \vdash_\Sigma p\hat{:}S, \Psi\ \mathsf{ok}}
$$

Henceforth, it will be assumed that all judgments are considered relative to a particular fixed signature Σ, and the signature indexing each of the other typing judgments will be suppressed.

$$
\frac{}{\Gamma \vdash \mathsf{type} \Leftarrow \mathsf{kind}}\ \mathsf{type}\mathbf{KF}
\qquad
\frac{\Gamma \vdash A \Leftarrow \mathsf{type} \quad \Gamma, u\!:\!A \vdash K \Leftarrow \mathsf{kind}}{\Gamma \vdash \Pi u\!:\!A.\, K \Leftarrow \mathsf{kind}}\ \Pi\mathbf{KF}
$$

$$
\frac{\Gamma \vdash A \Leftarrow \mathsf{type} \quad \Gamma \vdash B \Leftarrow \mathsf{type}}{\Gamma \vdash A \multimap B \Leftarrow \mathsf{type}}\ {\multimap}\mathbf{F}
\qquad
\frac{\Gamma \vdash A \Leftarrow \mathsf{type} \quad \Gamma, u\!:\!A \vdash B \Leftarrow \mathsf{type}}{\Gamma \vdash \Pi u\!:\!A.\, B \Leftarrow \mathsf{type}}\ \Pi\mathbf{F}
$$

$$
\frac{\Gamma \vdash A \Leftarrow \mathsf{type} \quad \Gamma \vdash B \Leftarrow \mathsf{type}}{\Gamma \vdash A \,\&\, B \Leftarrow \mathsf{type}}\ \&\mathbf{F}
\qquad
\frac{}{\Gamma \vdash \top \Leftarrow \mathsf{type}}\ \top\mathbf{F}
$$

$$
\frac{\Gamma \vdash S \Leftarrow \mathsf{type}}{\Gamma \vdash \{S\} \Leftarrow \mathsf{type}}\ \{\}\mathbf{F}
\qquad
\frac{\Gamma \vdash P \Rightarrow \mathsf{type}}{\Gamma \vdash P \Leftarrow \mathsf{type}}\ {\Rightarrow}\mathsf{type}{\Leftarrow}
$$

$$
\frac{}{\Gamma \vdash a \Rightarrow \Sigma(a)}\ a
\qquad
\frac{\Gamma \vdash P \Rightarrow \Pi u\!:\!A.\, K \quad \Gamma; \cdot \vdash N \Leftarrow A}{\Gamma \vdash P\; N \Rightarrow \mathsf{inst\_k}_A(u.\, K, N)}\ \Pi\mathbf{KE}
$$

$$\frac{\Gamma \vdash S_1 \Leftarrow \mathsf{type} \quad \Gamma \vdash S_2 \Leftarrow \mathsf{type}}{\Gamma \vdash S_1 \otimes S_2 \Leftarrow \mathsf{type}}\ \otimes\mathbf{F} \qquad \frac{}{\Gamma \vdash 1 \Leftarrow \mathsf{type}}\ 1\mathbf{F}$$

$$\frac{\Gamma \vdash A \Leftarrow \mathsf{type} \quad \Gamma, u\!:\!A \vdash S \Leftarrow \mathsf{type}}{\Gamma \vdash \exists u\!:\!A.\, S \Leftarrow \mathsf{type}}\ \exists\mathbf{F} \qquad \frac{\Gamma \vdash A \Leftarrow \mathsf{type}}{\Gamma \vdash {!A} \Leftarrow \mathsf{type}}\ !\mathbf{F}$$

$$\frac{\Gamma; \Delta, x \hat{:} A \vdash N \Leftarrow B}{\Gamma; \Delta \vdash \hat{\lambda} x.\, N \Leftarrow A \multimap B}\ \multimap\mathbf{I} \qquad \frac{\Gamma, u\!:\!A; \Delta \vdash N \Leftarrow B}{\Gamma; \Delta \vdash \lambda u.\, N \Leftarrow \Pi u\!:\!A.\, B}\ \Pi\mathbf{I}$$

$$\frac{\Gamma; \Delta \vdash N_1 \Leftarrow A \quad \Gamma; \Delta \vdash N_2 \Leftarrow B}{\Gamma; \Delta \vdash \langle N_1, N_2 \rangle \Leftarrow A \,\&\, B}\ \&\mathbf{I} \qquad \frac{}{\Gamma; \Delta \vdash \langle\rangle \Leftarrow \top}\ \top\mathbf{I}$$

$$\frac{\Gamma; \Delta \vdash E \leftarrow S}{\Gamma; \Delta \vdash \{E\} \Leftarrow \{S\}}\ \{\}\mathbf{I} \qquad \frac{\Gamma; \Delta \vdash R \Rightarrow P' \quad P' \equiv P}{\Gamma; \Delta \vdash R \Leftarrow P}\ \Rightarrow\Leftarrow$$

$$\frac{}{\Gamma; \cdot \vdash c \Rightarrow \Sigma(c)}\ c \qquad \frac{}{\Gamma; \cdot \vdash u \Rightarrow \Gamma(u)}\ u \qquad \frac{}{\Gamma; x \hat{:} A \vdash x \Rightarrow A}\ x$$

$$\frac{\Gamma; \Delta_1 \vdash R \Rightarrow A \multimap B \quad \Gamma; \Delta_2 \vdash N \Leftarrow A}{\Gamma; \Delta_1, \Delta_2 \vdash R\,\hat{}\,N \Rightarrow B}\ \multimap\mathbf{E} \qquad \frac{\Gamma; \Delta \vdash R \Rightarrow A \,\&\, B}{\Gamma; \Delta \vdash \pi_1 R \Rightarrow A}\ \&\mathbf{E}_1$$

$$\frac{\Gamma; \Delta \vdash R \Rightarrow \Pi u\!:\!A.\, B \quad \Gamma; \cdot \vdash N \Leftarrow A}{\Gamma; \Delta \vdash R\ N \Rightarrow \mathsf{inst\_a}_A(u.\, B, N)}\ \Pi\mathbf{E} \qquad \frac{\Gamma; \Delta \vdash R \Rightarrow A \,\&\, B}{\Gamma; \Delta \vdash \pi_2 R \Rightarrow B}\ \&\mathbf{E}_2$$

$$\frac{\Gamma; \Delta_1 \vdash R \Rightarrow \{S_0\} \quad \Gamma; \Delta_2; p \hat{:} S_0 \vdash E \leftarrow S}{\Gamma; \Delta_1, \Delta_2 \vdash (\mathsf{let}\ \{p\} = R\ \mathsf{in}\ E) \leftarrow S}\ \{\}\mathbf{E} \qquad \frac{\Gamma; \Delta \vdash M \Leftarrow S}{\Gamma; \Delta \vdash M \leftarrow S}\ \Leftarrow\leftarrow$$

$$\frac{\Gamma; \Delta; p_1 \hat{:} S_1, p_2 \hat{:} S_2, \Psi \vdash E \leftarrow S}{\Gamma; \Delta; p_1 \otimes p_2 \hat{:} S_1 \otimes S_2, \Psi \vdash E \leftarrow S}\ \otimes\mathbf{L} \qquad \frac{\Gamma; \Delta; \Psi \vdash E \leftarrow S}{\Gamma; \Delta; 1 \hat{:} 1, \Psi \vdash E \leftarrow S}\ 1\mathbf{L}$$

$$\frac{\Gamma, u\!:\!A; \Delta; p \hat{:} S_0, \Psi \vdash E \leftarrow S}{\Gamma; \Delta; [u, p] \hat{:} \exists u\!:\!A.\, S_0, \Psi \vdash E \leftarrow S}\ \exists\mathbf{L} \qquad \frac{\Gamma, u\!:\!A; \Delta; \Psi \vdash E \leftarrow S}{\Gamma; \Delta; !u \hat{:} !A, \Psi \vdash E \leftarrow S}\ !\mathbf{L}$$

$$\frac{\Gamma; \Delta \vdash E \leftarrow S}{\Gamma; \Delta; \cdot \vdash E \leftarrow S}\ \leftarrow\leftarrow \qquad \frac{\Gamma; \Delta, x \hat{:} A; \Psi \vdash E \leftarrow S}{\Gamma; \Delta; x \hat{:} A, \Psi \vdash E \leftarrow S}\ \mathrm{A}\mathbf{L}$$

$$\frac{\Gamma; \Delta_1 \vdash M_1 \Leftarrow S_1 \quad \Gamma; \Delta_2 \vdash M_2 \Leftarrow S_2}{\Gamma; \Delta_1, \Delta_2 \vdash M_1 \otimes M_2 \Leftarrow S_1 \otimes S_2}\ \otimes\mathbf{I} \qquad \frac{}{\Gamma; \cdot \vdash 1 \Leftarrow 1}\ 1\mathbf{I}$$

$$\frac{\Gamma; \cdot \vdash N \Leftarrow A \quad \Gamma; \Delta \vdash M \Leftarrow \mathsf{inst\_s}_A(u.\, S, N)}{\Gamma; \Delta \vdash [N, M] \Leftarrow \exists u\!:\!A.\, S}\ \exists\mathbf{I} \qquad \frac{\Gamma; \cdot \vdash N \Leftarrow A}{\Gamma; \cdot \vdash {!N} \Leftarrow {!A}}\ !\mathbf{I}$$